USENIX LARGE INSTALLATION SYSTEMS ADMINISTRATION IV CONFERENCE AUTUMN 1990

U

US

USE

U NIX

USENIX

CONFERENCE PROCEEDINGS

LARGE INSTALLATION SYSTEMS ADMINISTRATION IV

October 18 - 19, 1990
Colorado Springs, CO

For additional copies of these proceedings contact

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

The price is $15 for members and $18 for nonmembers.

Outside the U.S.A and Canada, please add
$8 per copy for postage (via air printed matter).

USENIX Association

Proceedings of the Fourth Large Installation System Administrator's Conference

October 17 - October 19, 1990
Colorado Springs, Colorado, USA

TABLE OF CONTENTS

AUTHOR INDEX

PROGRAM COMMITTEE

Steve Simmons
Industrial Technology Institute

Kevin C. Smallwood
Purdue University Computing Center

Elizabeth D. Zwicky
SRI International

Nick Simicich
IBM – T. J. Watson Research Center

REVIEWERS

Bob Filer
U. S. Air Force Academy

Rob Kolstad
Sun Microsystems

Evi Nemeth
University of Colorado, Boulder

Bjorn Satdeva
SysAdmin, Inc.

CONFERENCE ORGANIZERS

Steve Simmons, *Technical Program Chair*
(Industrial Technology Institute)

Judith F. Desharnais, *Meeting Planner*
(USENIX Association)

John Donnelly, *Tutorial Coordinator*
(USENIX Association)

Rob Kolstad, *Proceedings Layout*
(Sun Microsystems, Inc.)

Evi Nemeth, *Proceedings Production*
(University of Colorado, Boulder)

Trent Hein, *Proceedings Production*
(University of Colorado, Boulder)

LISA IV
October 17-19, 1990
Colo. Springs, Colorado

David A. Curry – SRI International
Samuel D. Kimery, Kent C. De La Croix, and Jeffrey R. Schwab – Purdue University

ACMAINT: An Account Creation and Maintenance System for Distributed UNIX Systems

ABSTRACT

ACMAINT is a network-based, centralized database system used to manage computer account creation and maintenance on the Purdue University Engineering Computer Network. ACMAINT allows the system administrator to perform account-related admintrative chores for any machine on the network from any attached system. Using ACMAINT, the system adminstrator can create new user accounts, add or delete accounts for existing users, change the global or per-account information associated with a user, place a message on a user's account(s), and enable or disable a user's account(s). Group information and mail aliases are managed in a similar fashion. ACMAINT utilizes a central database, stored on a single network machine, which contains a copy of all data under ACMAINT's control. The system administrator makes changes to the database via a network server running on the database machine, which in turn makes changes around the network via the use of another network server which runs on each machine. Programs which read, but do not write, the standard UNIX system databases such as the password file do not need to be modified to work with ACMAINT. Programs which write the standard databases must be modified or rewritten to converse with the ACMAINT database server. ACMAINT operates transparently to the user, uses minimal network and system resources, and can be used with binary-only UNIX systems.

Introduction

The Purdue University Engineering Computer Network[1] provides computing services to the seven Schools of Engineering using a network of approximately 25 UNIX timesharing machines and over 300 Sun workstations. Across the whole network, there are about 12,000 individual users and over 90,000 actual accounts. Because most of the computer accounts on these systems belong to students, there is a large amount of turnover each semester. In the School of Electrical Engineering alone, nearly 2,000 new accounts are created each fall, and between 1,000 and 2,000 accounts are deleted each summer.

Each School of Engineering has administrative control over the machines it owns. A system administrator in each school has the responsibility for creating, deleting, and updating computer accounts for the students, faculty, and staff in that school. The system administrators must coordinate their activities with each other in order to avoid creating problems. The large turnover in accounts presents several problems both in account creation (duplicate login names, etc.) and account deletion (deleting users with accounts on more than one department's machines, etc.).

[1]This work was done while David Curry was employed by the Purdue University Engineering Computer Network.

Other problems surfaced as the network expanded to include more and more workstation systems. Changing information for a user with many accounts (e.g., an expiration date or classification) was a time-consuming process which required the system administrator to log in to each machine the user had an account on and change the information manually. This was particularly problematic at the start of each fall semester, when many users who were classified as "freshman engineering" became sophomores and selected a specific discipline, requiring a change in their classification.

The problems of managing unique user ids and login names, time-consuming account installation and maintenance processes, and the desire to have all user information centrally located convinced us that a system to manage computer account creation and maintenance on a network-wide basis was needed. The system selected had to meet several requirements, including:

- Centralized data storage
- Machine and vendor independence
- Flexibility in data to be stored
- Minimal changes to existing software
- Automated account installation
- Easy recovery from crashes
- Automated account deletion
- Simultaneous access for multiple users

In addition to these requirements, several other features were identified as ''nice to have'' but not required. These included the ability for the system administrator to change any piece of information from any host, the ability to change a piece of information (e.g., a user's password) on all hosts with a single command, the ability to generate ''batch'' updates to the database (e.g. from a Registrar's tape), and the ability to search the database for various combinations of information.

Previous Approaches to the Problem

Before deciding to implement our own account maintenance system, we examined other approaches to the problem of large network system administration. We had hoped that someone who had already faced our problems would have solved them to our satisfaction. Unfortunately, this was not the case. This section briefly describes the three systems we examined, and our reasons for not using them.

Yellow Pages

Yellow Pages (YP) [4] is a distributed network lookup service from Sun Microsystems. It is perhaps the most widely-known approach to system administration on a large network. YP maintains a set of files, called *maps*, which contain keys and associated values. YP maps can be replicated on several systems known as YP servers, each of which runs a server process for the maps. C library routines which query UNIX databases such as the password or hosts files are modified to query a YP server instead of reading the file directly. This enables the system administrator to make changes to only a few files (the ones on the YP master machines) and have the changes take effect on all the machines in the network. Programs require no modifications to use YP; when they are linked with the new C library they automatically gain this capability.

YP meets most of the requirements set forth in Section 1: it centralizes data storage, is flexible in what data may be stored, is machine and vendor independent, and requires no software changes. Unfortunately, there are no provisions for automated account installation or deletion, which we required. Also, because the YP maps are built from the standard UNIX databases, there are no provisions for allowing multiple users to modify the database simultaneously.

It may be possible to build utilities which provide these capabilities on top of YP, but we discovered several problems with YP which make it undesirable. First, using YP means significantly increasing the amount of network traffic generated by each machine on the network. Because database accesses are now turned into network accesses, even a simple **ls -l** command generates a large amount of traffic. A related problem involves the updating of YP maps on secondary (slave) servers: whenever a map is updated, the *entire map* is transferred over the network to each slave server. Thus, one user changing his password can result in several kilobytes of network traffic. Still another problem comes from the fact that although databases are maintained by YP, some of them must still reside on each machine as plain text files. These files must be updated on a regular basis in order to avoid becoming out of date with respect to the YP maps. Perhaps the most serious problem though, is that if the YP servers become unreachable for any reason (the server machine crashes, the network routing becomes confused, etc.), any program which uses YP will hang. This would not be a problem if only rarely used commands were affected, but a YP server failure affects even such simple commands as **ls**. Thus, the failure of a single machine could easily affect hundreds of others. For these reasons, Yellow Pages was rejected as a solution to our problem.

Athena Service Management System

The Service Management System (Moira) [2], developed by M.I.T.'s Project Athena, maintains a centralized database of user information, as well as other data, using Ingres from Relational Technology, Inc. A Moira process reads the database each night and generates password files, group files, and so on. These generated files are then transferred to the hosts they correspond to, and installed by another process after several consistency and validity checks have been made. Moira, as of February, 1988, handled 15,000 accounts and 1,000 machines.

Moira also meets most of the requirements listed in Section 1: it centralizes data storage, is flexible in the data that can be stored, provides automated account installation (but not complete account deletion), is machine and vendor independent, and allows the database to be updated simultaneously by multiple users. The system can recover from crashes, but this is done by loading ''plain text'' backup files (made nightly) back into the database (which brings it current with the last backup), and then by manually consulting log files and redoing any transactions which took place since the last backup.

The problems which we found with Moira were not as serious as those found with Yellow Pages, but were still serious enough to force us to reject Moira as a solution to our problem. One problem with Moira is that because it generates files on one machine and then propagates these files to their proper locations, machines can become out of date if they are down during the transfer. This is remedied by running the transfer several times during the day (checks are made to avoid repeated updates), but this is still not completely desirable. A related problem is that transferring complete files across the network generates a lot of excess network traffic, although much of this traffic is only generated late at night. Finally, the problem which most turned us against Moira was a financial one. Moira uses the Ingres database system from Relational Technology, Inc. This database system is very expensive, as are others such as Unify and Oracle, which could also be used. An experiment with "University" Ingres convinced us that for performance reasons, a commercial database product is required to implement Moira.

Asmodeus

Asmodeus [1], developed at Oregon State University, maintains a database of all user information, and provides access to that database through a series of *database daemons*. All changes to the database are made through these database daemons, as well as all database accesses. When a change is made to some part of the database, the database daemon sends commands to one or more *activity daemons*, which run on each network machine. These activity daemons then make changes in the password file, group file, or whatever. For example, when a user changes his password, a command is sent to a database daemon, which makes the change in the central database, and then sends a command to the activity daemon on the user's machine. The activity daemon then edits the regular UNIX password file, making the change complete. This approach has two primary advantages: changes take effect immediately rather than overnight, and programs which read the UNIX databases do not need to be modified since the use of the activity daemons preserves those files.

Unfortunately, the design of Asmodeus presents a few problems which, while not insurmountable, were in our view unacceptable. The most problematic part of Asmodeus is the use of multiple database daemons. Rather than having a single copy of the database, Asmodeus maintains multiple copies, with multiple database daemons. These daemons communicate among themselves in order to keep the multiple copies of the database current. In our view, the multiple daemons complicated matters by requiring database records to be locked before being changed (to prevent two different daemons from changing the same record), and so on. The algorithm for deciding when a database daemon was up or down was also deficient; a segmented network could easily result in two or more copies of the database being changed in different ways at the same time. Asmodeus uses a reliable datagram protocol, which we felt was more complex than necessary. By using a simple send-and-acknowledge scheme, the overhead of a reliable datagram protocol can be avoided. These problems in themselves would not have prevented us from opting to use Asmodeus rather than designing our own system. The database concurrency problems could be solved by running only a single database daemon, and the reliable datagram protocol could easily be removed.

Asmodeus is perhaps the closest match to what we wanted, and in fact much of ACMAINT's design is at least loosely based on it. If it hadn't been for the fact that, at the time we were looking for available systems, coding for Asmodeus had only just begun, ACMAINT would probably not exist today. Asmodeus fits all of the requirements outlined in Section 1, as well as making most of the "nice to have" features available or easily implementable.

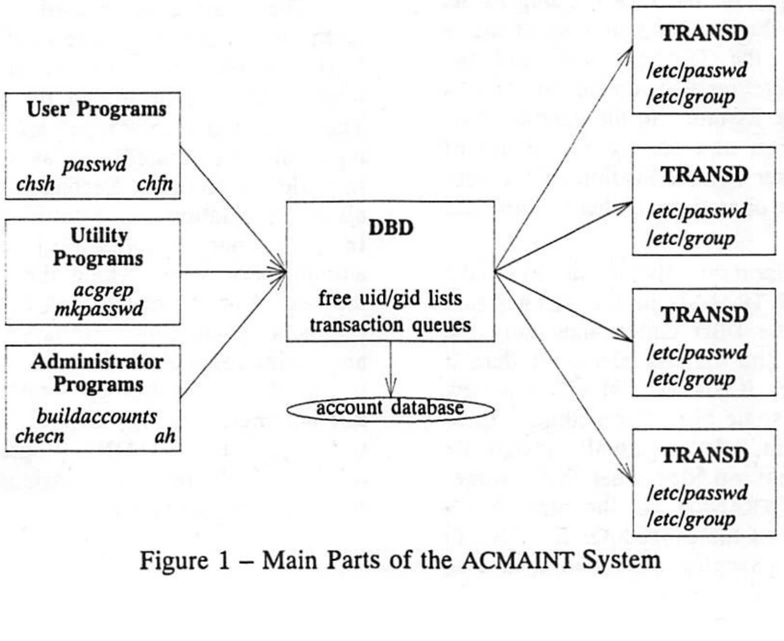

Figure 1 – Main Parts of the ACMAINT System

ACMAINT Design and Implementation

The design of ACMAINT satisfies both the primary requirements and the ''nice to have'' features of an account management system as described in Section 1. The storage of data about users, accounts, and groups is centralized on a single network host. A network server is provided to access the database, which allows changes to be made to the database from any host on the network. The use of other network servers, one on each network host, allows the installation and deletion of accounts, as well as the changing of account-related information, to be automated. ACMAINT's design divides these features into three main parts: the database daemon, the transaction daemon, and programs for users and system administrators. Figure 1 shows the relationship between these parts of the system.

The database daemon (DBD) is a network server which resides on the network host where the central database is stored. It is responsible for making all database retrievals, additions, deletions, and changes. The DBD accepts UDP datagrams containing database manipulation commands from other ACMAINT programs, acts on them, and returns the results in other datagrams. After performing any database updates required by a command it receives, the DBD constructs commands which are sent via UDP datagrams to the transaction daemons on the affected hosts, where changes to the appropriate UNIX database files are made in order to complete the command.

The transaction daemon (TRANSD) is a network server which resides on every network host under ACMAINT's control. It is responsible for installing and deleting accounts and groups, creating, moving, and deleting user's home directories, disabling or enabling accounts, and so on. The TRANSD accepts UDP datagrams containing commands sent from the DBD, and makes changes to the files residing on its host. For example, when a command to create a new account arrives, the TRANSD will add the appropriate line to */etc/passwd*, create the user's home directory, change its mode to the default mode for new user directories, and change the owner of the directory to the user. An indication of the success or failure of these operations is then returned to the DBD.

There is an important distinction to make between the DBD and TRANSD in the manner that they process data. The DBD understands only the *relationships* between the various pieces of data it handles. For example, it knows that when a new account is created on some host, ''something'' must be done on that host in order to actually install the new account, but is has no idea what that ''something'' may be. The TRANSD, on the other hand, has full understanding of the procedures involved in installing an account, changing a password, and so on. This distinction is important, because it allows ACMAINT to control hosts running any number of different UNIX variants. Because the DBD does not need to know exactly how to install an account, it does not need to know whether a host uses simply */etc/passwd*, the Berkeley *passwd.dir* and *passwd.pag* files, or even some ''shadow'' password file scheme. The TRANSD on each of these hosts will know the procedures required to install an account on that machine.

User-level account-related programs interface to the ACMAINT system via the DBD. Any program which modifies one of the account-related UNIX databases, such as **passwd**, **chfn**, **chsh**, etc., has been modified to instead send commands to the DBD. The DBD accepts these commands, modifies the database, and then sends commands to the TRANSD on the user's host. The TRANSD then modifies the file(s) which were previously modified directly by the program run by the user.

Similarly, new tools have been written for the system administrator that allow accounts to be created, deleted, and changed. The principal program used for all this is **ah** (Account Handler), which can function either in an interactive fashion or by reading ''batch'' files of commands. **ah** is described in more detail in a later section

Database Format

The ACMAINT database is implemented using a version of the Berkeley **ndbm** [5] library which has been modified to allow storage of larger records. The **ndbm** routines are not called directly; ACMAINT library routines which encode and decode database records to and from the format used in the database are used instead. This organization allows the easy substitution of some other database mechanism, such as the **gdbm** library from the GNU project or a commercial database management system.

There are five record types stored in the ACMAINT database. Three of these record types are ''primary records,'' that is, they pertain to data which will be reflected in the UNIX database files. The other two record types are ''secondary'' record types, and are created only as a side effect of creating primary records. Secondary records are used to allow information to be retrieved using other keys. In all primary records, the name of the system administrator who created the record and the time the record was created are maintained to provide rudimentary auditing. Primary records also contain an ''uninterpreted data'' field which is stored as a list of data separated by semicolons. This field is not interpreted by the DBD or TRANSD, but can be used by other ACMAINT programs which read the database and create permissions files and the like from information stored here.

The *user record* is a primary record which stores information that is unique to a user, but constant across all accounts the user may have. It is stored in the database using the user's login name as the key. There is one user record in the database for each person with an account on any of the systems managed by ACMAINT. Creation of a user record does not cause any UNIX database files to be modified. The user record contains:

- Numeric user id
- Login name
- Student id number
- Full name
- Office room number
- Office phone
- Home phone
- Mailbox address
- Group memberships
- Departmental affiliations
- Creation time
- Created by
- Uninterpreted data

The *account record* is a primary record which stores the information about a user which is not constant across all the user's accounts. It is stored in the database using the key ''*login@host*''. There is one account record for each account a user has. Creation of an account record causes a command to be sent to *host*'s TRANSD requesting that an account be created. The account record contains:

- Host name
- Login name
- Login group id
- Password (encrypted)
- Classification
- Courses
- Authorizing department
- Authorizing person
- Expiration date (mo/yr)
- Login shell
- Home directory
- Creation time
- Created by
- Uninterpreted data

The *group record* is a primary record which stores information about a group. It is stored in the database using the key ''*groupname*%group''; this allows group names to be the same as login names where necessary. Creation of a group record causes a command to be sent to all TRANSDs requesting that the group be created. The group record contains:

- Numeric group id
- Group name
- Password (encrypted)
- Authorizing person
- Number of members
- Creation time
- Created by
- Uninterpreted data

The *host record* is a secondary record which is used to maintain a list of all hosts a user has an account on. It is stored in the database using the key ''*login*%hostlist''. The contents of this record can be used to generate the keys required to retrieve all of a user's account records. This record is modified whenever an account record is created or deleted.

The *student id record* is a secondary record which is used to allow a user's login name to be determined given his student id number. It is stored in the database using the key ''*studentid*%sid''. This record is used primarily to allow detection of duplicate accounts for the same student, which ACMAINT will not permit (a local policy which was designed into the software).

The Database Daemon

When the DBD begins execution, it clears its controlling terminal, creates a UDP socket at a well-known port to receive packets, and opens the database. Next, the on-disk queue of unacknowledged packets (see below) is loaded so that the packets can be retransmitted. Finally, the DBD goes into an infinite loop of receiving and processing packets, and retransmitting packets which were not acknowledged.

When a packet is received, it is first examined to insure that it was sent by an authorized sender. This check presently consists only of making sure the packet came from a privileged port on the sending machine. Assuming the packet is legitimate, it is then processed. If the packet came from a host which the DBD thinks is ''down,'' the host is recorded as now being ''up.'' If the packet is an acknowledgement packet sent by a TRANSD, it is removed from the in-memory and on-disk queues of unacknowledged packets. If the packet contains a command, control passes to the function responsible for implementing that command.

A command is processed by retrieving the necessary data from the database and if necessary, modifying the data and storing it back in the database. A command packet is then constructed for each TRANSD affected by the command (if any), and queued for transmission to that TRANSD after any other pending traffic has been sent. The packet is also stored in a queue of unacknowledged packets for possible retransmission. Next, an acknowledgement packet (or error packet, if the command failed) is constructed and sent back to the sender of the command. The acknowledgement packet may contain data; for example, the packet containing the results of a ''fetch'' operation also serves as an acknowledgement of the command. The acknowledgement packet is stored in a queue of ''recent'' acknowledgements, in order to allow detection of duplicate command packets. If another command packet arrives which is a duplicate of this one (i.e., the client program retransmitted it because no acknowledgement was received), the acknowledgement packet is simply retransmitted, without performing the command a second time. This is important, since many commands such as deleting an account are not repeatable.[2]

As mentioned above, unacknowledged packets are stored in a queue for later retransmission. This queue is maintained in memory for immediate use by the DBD, and also on disk to preserve state across machine or DBD crashes. Each time through the main receive/retransmit loop, if there are packets to be retransmitted, the DBD retransmits one packet. When a packet is retransmitted, the host it is sent to is marked as "down" by the DBD. If more than 20 minutes have passed since the host was marked "down," or if a new packet is received from the host, it is marked "up" again. This enables timely updates to hosts which have crashed without flooding the network with packets destined for hosts which are not responding. Packets are retransmitted in a simple round-robin fashion, although this algorithm can easily be changed if delays between retransmissions become too large.

The commands shown in Figure 2 may be sent to the DBD by an ACMAINT client program. Some commands, such as "create_group", allow some fields to be specified as "*", which instructs the DBD to select a unique value for that field. Any command which uses the syntax "login@machine" will allow the machine to be specified as "*", indicating that this command should be performed on all machines on which the user has an account.

[2]Commands are generally "benignly" non-repeatable: an accidental reissuance of a non-repeatable command will result in a bogus error message, but will not cause any permanent damage.

The Transaction Daemon

The TRANSD is started via **inetd** [6], in order to conserve resources on machines without a lot of memory (e.g. the Sun 3/50). When a packet is received, it is first checked for authenticity. This check currently consists of making sure the packet was sent from the host and port used by the DBD. The only type of packet a TRANSD expects to receive is a command packet. Other packet types (errors, acknowledgements, etc.) are logged and discarded. If the packet is a command packet, control is transferred to the function which implements that command.

A command is processed by modifying the appropriate UNIX database files (*/etc/passwd*, */etc/group*, etc.) and by creating or removing any necessary files and directories. An acknowledgement packet (or error packet, if the command failed) is then constructed and sent back to the DBD. The acknowledgement packet is stored in a queue of "recent" acknowledgements, in order to allow detection of duplicate command packets. If another command packet arrives which is a duplicate of this one (i.e., the DBD retransmitted it because no acknowledgement was received), the acknowledgement packet is simply retransmitted, without performing the command a second time. This is important, since the command may not be repeatable (e.g., changing a login name).

```
add_group gname login                                  enable login[@machine]
change_acct login@machine field value                  fetch_acct login@machine [field]
change_group gname field value                         fetch_group gname [field]
change_user login field value                          fetch_hosts login
delete_acct login@machine                              fetch_user login [field]
delete_group gname                                     message login[@machine] message
delete_user login                                      remove_group gname login
disable login[@machine] message                        unmessage login[@machine]
create_acct login:machine:gid:passwd:classif:authdept:authorizer:\
          expdate:shell:homedir:courselist:uninterp
create_group gid or *:gname:passwd:authorizer:uninterp
create_user uid or *:login or *:sid:fullname:office:offphone:homephone:\
          mailbox:grouplist:affiliationlist:uninterp
```

Figure 2 – DBD Commands

```
add_group gname login                      delete_group gname
change_acct login field value              disable login message
change_group gname field value             enable login
create_acct password-file-line             message login message
create_group group-file-line               remove_group gname login
delete_acct login                          unmessage login
```

Figure 3 – DBD Commands to the TRANSD

The commands shown in Figure 3 may be sent to the TRANSD by the DBD. In general, these commands correspond to the DBD commands of the same name; only the arguments to the commands are different.

ACMAINT Client Programs

A large number of client programs have been written or planned in order to allow system administrators and ordinary users to access and change the information maintained by ACMAINT.

The Account Handler Program

The *account handler* program, **ah**, is the major ACMAINT interface for the system administrator. Using **ah**, the system administrator has access to all DBD commands. Commands can be executed interactively via prompts, or "batch" scripts can be created (either manually or by other programs) and used as input. The present interactive interface is line-oriented, although a screen- or window-oriented interface could easily be written if desired. The current version of **ah** is actually just an exceedingly clever shell script with a couple of "helper" programs for sending packets and verifying field contents.

One of the most useful features of **ah** is the *.ahrc* file. This file allows the system administrator to specify default values for ACMAINT database record fields so that they do not need to be specified for every command. For example, appropriate defaults can be set for the user's department affiliation, classification, and expiration date. Default values can be overridden simply by specifying a value for the field in question. The *.ahrc* file also allows the system administrator to define arbitrary variables which can then be substituted into **ah** commands using the shell's "$" syntax. For example, a variable can be defined which contains the names of all clients of a specifc file server, and then a single command can be used to add an account to all of those hosts.

Other System Administrator Programs

Other ACMAINT system administrator programs which have been written so far include a program which reads a password file and translates it into ACMAINT database records (used for placing new hosts under ACMAINT's control), a program which reads the database and generates a password file for any host in the database, and a similar program which reads the database and generates a group file.

Several other programs are under development or planned for the future. These include:

- **acgrep**, a program which allows the database to be searched using regular expressions, logical expressions, and relational expressions. For example, the database can be searched for all accounts on host "harbor" that expired on or before May, 1989 and print a list of login names:

```
acgrep -host harbor && \
        -expdate <= 0589 -print login
```

- a program to read a tape provided by the Registrar which contains the names, student id numbers, and graduation dates of all enrolled students and generates an **ah** batch script to create accounts for all new students and disable accounts for students who are no longer enrolled,
- a program to search the ACMAINT database for accounts which are about to expire and place messages on those accounts telling the user that the account will be disabled and deleted once it expires (this can be a shell script built on top of **acgrep**),
- a program which searches the database for accounts which have expired and generates both a shell script to place the files owned by those accounts onto tape and an **ah** batch script to delete the accounts (this can also be a shell script built on top of **acgrep**),
- **buildaccounts**, a front end to **ah** which has a screen-oriented interface and extensive error checking,
- **lahbel**, a program which reads the ACMAINT log files and generates "sticky labels" containing the login names and passwords of newly created accounts. These labels are then stuck to the forms which explain how to use the system, where to find documentation, etc. and given to the users.

The design of ACMAINT is extremely flexible and allows the development of all sorts of utility programs. As our experience with ACMAINT grows, other ideas for useful programs are sure to come up.

Modifications to User Programs

User-level programs which modify UNIX databases (**passwd**, **chfn**, and **chsh**) have been modified (or simply rewritten) to converse instead with the DBD. These modifications essentially involve replacing all the code involved in modifying the UNIX database file with about 25 to 50 lines of code calling various ACMAINT library functions to send and receive packets to and from the DBD. The wildcarding allowed by ACMAINT on database "change" commands has also allowed us to add a "network wide" option to these commands. By specifying the "–n" option to **passwd** or **chsh** ("–n" is implicit for **chfn**), the user can request that the information be changed on all of his or her accounts.

Life With ACMAINT

ACMAINT was initially brought up for testing in April, 1989 and controlled approximately 200 hosts. As of August, 1989 it controlled all hosts in the Engineering Computer Network, a total of over 300 machines. It is regularly used by seven system administrators, as well as by users changing their passwords, etc. The DBD is presently running on a Sun 4/280 server with 32MB of memory and is providing very good response time. The database consumes about 22MB of disk space, which averages out to approximately 75KB per host maintained. The TRANSD is running on VAX-11/780s and CCI Power 6/32s under 4.3BSD, Gould NP-1s and PN9080s under UTX/32 2.0, and Sun 3/50s, 3/60s, 3/180s, 4/280s, and 4/390s under SunOS 4.0.3. At present, we are using ACMAINT Version 1.4.

Our system administrators are generally pleased with ACMAINT, although they are anxious to have the programs mentioned in the previous section. They report that the time required to install new accounts (the most time consuming task at the start of a semester) has been cut in half, the time required to move a user's directory from one file system to another has been reduced to almost nothing (simply use **ah** to change the home directory in the account record, and let the TRANSD move the directory), and the account deletion process has been greatly simplified since the whole procedure is now automated.

It is expected that as more auxiliary ACMAINT software is developed, the amount of time spent by the system administrator on account-related tasks will decrease even more. The number of mistakes made in installing accounts (forgetting to change the ownership of the home directory, typos in the password file, etc.) has already been greatly reduced due to the automation and error checking provided by ACMAINT.

Portability

ACMAINT was written with portability in mind, since all programs except the DBD must work on a wide variety of systems. At present, no attempt has been made to port it to any versions of UNIX not based on 4.2 or 4.3BSD. However, provided a Berkeley socket interface is available, porting should not prove difficult. Care has been taken to make ACMAINT as configurable as possible, so that it can work on as many different systems as possible without writing more code.

Porting the TRANSD to a new variant of UNIX involves modifying the code which implements each TRANSD command. This includes the proper procedures to update the password file, create an account, modify the group file, and so on. Most systems should be similar enough to the Berkeley or Sun variants that little modification will actually be necessary.

Making modifications to **passwd** and other programs is probably the most problematic part of porting ACMAINT. This is because source code is required to make the modifications, but most sites have binary-only systems. In the future, we hope to provide "generic" ACMAINT implementations of these commands which will be suitable for use on most systems.

Problems Encountered

Even after extensive testing before putting ACMAINT into use, we were unfortunate enough to run into several problems, mostly centered around synchronizing machines which had gotten out of date somehow.

Initially, getting error messages from a TRANSD back to the user (going through the DBD) was difficult, and many messages got lost. This has since been fixed by having the DBD keep more state information about commands it has sent to TRANSDs. In the next version of the system, error messages will be sent back to the user via electronic mail.

Another problem became noticeable once a large number of machines were placed under ACMAINT's control: because the DBD sent commands to the TRANSDs immediately, some "network wide" commands (e.g. changing the password for "root") would cause the DBD to become unresponsive for several minutes until all the packets were sent. This problem was solved by sending acknowledgements for these commands after the modification of the database but before all the TRANSDs had been contacted. (This is safe, since all TRANSDs must still acknowledge completion of the command to the DBD.)

Problems have also started to arise with the round-robin retransmission scheme used by the DBD. The algorithm works, but because there are so many machines under ACMAINT's control, delays between updates are starting to grow to unacceptable lengths. This will be fixed in the next version of the system by splitting the single retransmission queue into separate queues for each host. The whole back-off/retry scheme presently in use also needs a major overhaul, and will probably be fixed by going to some sort of per-host bounded exponential back-off scheme.

Future Directions

In the next version of ACMAINT (Version 2.0), we plan to make several changes and additions, including:

- Kerberos [3], from M.I.T.'s Project Athena, will be added for authentication between client programs and the DBD, and between the DBD and TRANSDs.
- Support in the TRANSD for the Berkeley "shadow" password file mechanism, which will be in the next version of Berkeley UNIX.
- The functions of the DBD will be split and provided by three cooperating processes:

DBD	Talks to clients and manages the database. Writes a single work record to disk, and signals the BURSTD.
BURSTD	Separates a work record written by the

DBD into individual packets and places them into multiple packet queue directories, and then signals the WIRED.

WIRED Handles communications from the DBD to the remote TRANSDs. Each remote machine has its own packet queue directory, and transmit speed is limited by the rate at which the remote machine can complete requests. A bounded exponential back-off scheme is used for retransmissions. This provides higher throughput and lower memory usage. Splitting the DBD functions into three processes allows the DBD to respond to user requests rapidly, leaving the time-consuming tasks of splitting work requests into packets for the TRANSDs and sending the packets to the remote machines to other processes.

Availability

When it is completed, we plan to make ACMAINT Version 2.0 available to the UNIX community at large via anonymous FTP and UUCP. It will also be posted to one of the USENET source groups. This will probably occur sometime in the fourth quarter of 1990.

We also have been carrying on discussions with Berkeley about including ACMAINT with the next release of Berkeley UNIX (following 4.3-Reno).

References

[1] M. E. Epstein, C. Vandetta, and J. Sechrest, "Asmodeus: A Daemon Servant for the System Administrator," in *USENIX Conference Proceedings*, San Francisco, Summer, 1988, pp. 377-391.

[2] M. A. Rosenstein, D. E. Geer, Jr., and P. J. Levine, "The Athena Service Management System," in *USENIX Conference Proceedings*, Dallas, Winter, 1988, pp. 203-211.

[3] J. G. Steiner, C. Neuman, J. I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," in *USENIX Conference Proceedings*, Dallas, Winter, 1988, pp. 191-203.

[4] Sun Microsystems, *System and Network Administration*, Part No. 800-1733-10, May, 1988, pp. 349-387.

[5] *UNIX Programmer's Reference Manual, 4.3 Berkeley Software Distribution*, Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, April, 1986.

[6] *UNIX System Manager's Manual, 4.3 Berkeley Software Distribution*, Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, April, 1986.

Dave Curry discovered "adventure" on a PDP-11 running UNIX V6 in 1977 while still in high school. Since then he has worked for Purdue University, the Research Institute for Advanced Computer Science, and SRI International. He is the author of *Using C on the UNIX System* from O'Reilly & Associates, and recently wrote the popular white paper, *Improving the Security of Your UNIX System*. Dave can be reached at SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025 or electronically at davy@itstd.sri.com.

Sam Kimery has been involved with Unix since 1980. Prior to his employment at Purdue University, Sam was a systems programmer at Ford Aerospace, and also served time as a consultant. He has been employed as a programmer with the Engineering Computer Network at Purdue University since 1987. Sam can be reached at Purdue University, Electrical Engineering Building, West Lafayette, IN 47907, or electronically at kimery@ecn.purdue.edu.

Kent C. De La Croix recieved his Bachelor of Science degree in Computer Science from Purdue University in 1979. He has worked as a systems admistrator and computer programmer. He has been employed as a programmer with the Engineering Computer Network at Purdue University since 1984. Kent can be reached at Purdue University, Electrical Engineering Building, West Lafayette, IN 47907, or electronically at kcd@ecn.purdue.edu.

Jeff Schwab was exposed to UNIX V6 on a PDP-11 in 1975. He survived long enough to complete an M.S. in Computer Science at Purdue University. After stints at both IBM and Dec, Jeff is now back at Purdue as the Network Services Manager for the Engineering Computer Network. Jeff can be reached at Purdue University, Electrical Engineering Building, West Lafayette, IN 47907, or electronically at jrs@ecn.purdue.edu.

LISA IV
October 17-19, 1990
Colo. Springs, Colorado

UDB – User Data Base System

Roland J. Stolfa and Mark J. Vasoll –
Oklahoma State University

ABSTRACT

UDB is a central database running on a UNIX system that maintains a universal, flat user name space for participating hosts. Though the UDB database itself resides on a UNIX system, it can manage user information for other username/password oriented systems as well (VAX/VMS, MVS/360, etc.). This document describes the operations and services provided in this system as implemented at Oklahoma State University.

Background

Prior to the development of UDB (the User DataBase), the two support staff members here at Oklahoma State University Computer Science Department had to manually maintain over 40 machines. This included all the normal system administration of hardware, backups, mail, and news, as well as account creation. We often found ourselves running around the department in search of which machine a user wanted an account on, logging on, creating the account, then finding the person and informing them of the initial password. In many instances, several days might pass between the request and the creation of the account due, in part, to playing "telephone tag". Another scenario would involve the creation of enough accounts for an entire class. This typically implied that there were no initial passwords for those students.

As time passed, access to the two terminal labs that the department runs for graduate students (and for special classes) also became a chore. This involved keeping track of many keys to the labs in question. Monitoring the labs, and the equipment contained therein, became enough of a worry that a set of magnetic strip readers and a card key access system was installed on six doors in the department to monitor the use of the rooms.

With the addition of NFS (Network File System) hosts/servers to the hardware mix of the department, file sharing amongst one user's account on several machines became a problem due to NFS needing the numeric user id's to be the same on all systems. This involved merging many separate password files, sorting out what id's were in use on what machine, and then changing the numeric user id's across the network.

In short, the job for the two support staff members became a 40+ hour a week job with little or no free time to devote to installing new software packages, patching old ones etc. In self defense, we wrote UDB.

Introduction

UDB is a collection of programs, running on a UNIX system, that maintains a central database of user information. It currently provides an automatic updating, triggered by the central university enrollment database, for creation of instructional accounts, generalized update messages to all hosts with add/delete/change information for each host, update information to a card key access security system, extraction of system wide authorization data, and generation of modified class roll information for use by instructors.

Although the overall functionality has not changed significantly since the first implementation, many refinements have occurred. This document will attempt to cover how the current system is designed and behaves.

Services Provided

The UDB system provides to participating hosts a file that contains selected fields of the database for all users that are selected for that host. For each destination operating system, a specialized program (on the server) generates the type of authorization file needed by the account generation software on the destination host (client). This software is then responsible for taking the data presented and doing something with it. As each operating system is different, the methods used to decide if a user is new, changed, or deleted must be customized for that system. For the UNIX systems within the Computer Science Department, the program "upd_unix.sh" does the job, and should be taken as the model for user account addition, modification, and deletion for any new interfaces for different operating systems.

It should be noted that when UDB sends an authorization file to a destination host, it is up to the client host administrator to decide the level of action to be performed next. The system administrator may initiate the commands to actually update the current user list on that host to reflect what UDB indicates it should be. Alternatively, the system administrator may configure the system to automatically take the data and update the current user list on that host.

Although UDB was intended to be the sole authority as to which users would be on a client machine, this decision is also left up to the client system administrator. A system administrator may treat the data from UDB as supplementary, allowing the UDB software to maintain certain accounts, while retaining local accounts that are outside of UDB's domain. Either of these options is accomplished in the Unix case by a site specific configuration file that defines the value of the numeric user id's above which UDB is allowed to create and remove accounts.

By way of example, the list of services provided to hosts within the OSU Computer Science Department that use this system is outlined below:

- An authorization file that contains the Universal Computing Identifier (UCI), encrypted password, the full name of the user, the list of automatic rights, and the list of granted rights for every user of a particular host.
- The configuration data for a card key lock system that audits access to two computer labs and two machine rooms within the Computer Science Department.
- Class enrollment lists that give the student's university identification number, the full name field (as generated by the university), the UCI (as generated by UDB), and the clear text password (as generated by UDB).
- Mailing lists for each of the selection criteria that was used to select users for this system. This typically includes a mailing list for each class taught on a particular system.
- An authorization file that contains all of the rights of all users of the system, in summary.

The Database

Conceptually, the database is a relational database. The database contains the following pieces of information about our users as one logical record:

```
Student/Faculty identification number (ID)
Student/Faculty ID card issue number
     (ISSUE)
Full name as defined by the university
     (FULLNAME)
Universal Computing Identifier (UCI)
Preferred Numeric User ID for NFS (NUID)
Clear text initial password (PASSWD)
Encrypted initial password (CPASSWD)
Student department affiliation (MAJOR)
Automatic rights (AUTO)
Granted rights (GRANTED)
Last update time of this record (LUPDATE)
```

Due to the operational use of these fields however, the logical record is divided into three physical records that are described below. Each of these records is indexed by the student/faculty identification number.

```
General Record
     ID, ISSUE, FULLNAME, UCI,
     NUID, PASSWD, CPASSWD, MAJOR,
     LUPDATE
Automatic Rights Record
     AUTO
Granted Rights Record
     GRANTED
```

In addition, there are two flag record types that are used as secondary indexes into the database. All of these fields are fairly self explanatory, except the concepts of automatic rights, granted rights, and flag records; thus, these are summarized below.

Automatic Rights Record

The data held in the automatic rights physical record are related to enrollment. Contained in this physical record is a list of all relevant courses this user is enrolled in this semester. All data in this field expires automatically when the new data from the central university data base is received.

These rights take the form of a comma separated list of text strings. This field of the logical record was split off into a separate physical record because the data contained in it would need to be updated each time a new course enrollment data feed was received. Having this as a separate field simplifies the update process by allowing the system to simply remove all automatic records in one pass. Then, the program that receives the enrollment data can simply reload the course enrollment data by regenerating all the automatic records.

Granted Rights Record

The data held in the granted rights physical record are special case rights given on an as needed basis. Such things allow users such as "root" to have an account on all machines without worry that their accounts will be automatically deleted at some point in the future. In addition, there are always special case users who need an account (or access to a room via the card key access system) for a specified period of time that would not otherwise be granted that right.

These rights take the form of a comma separated list of text strings where each right may have an expiration date of the form "–YYYYMMDD" appended to it. Upon a periodic scan of the database, all manual rights that have reached their expiration date are removed from that user's record.

This field of the logical record was split off into a separate physical record because it was found that there were very few of these compared to the number of total users in the database (10:1 on average).

Flag records

The two flag records are index pointers from one type of data to the student/ faculty identification number. They were used to improve the overall efficiency of the UDB system by allowing one database probe to determine the state of an index key by returning success or failure. The two instances in UDB are described below.

UCI mapping

In order to allow for a UCI to student/faculty identification number mapping, a physical record was inserted that performed this mapping. This also simplified the job of finding an unused UCI when a new student/faculty member came into the department. In one probe, the UDB system can determine if a particular UCI is in use or not.

NUID mapping

In order to maintain all NUID's as separate and unique, a method of determining which NUID's were in use was needed. In order to keep from sequentially scanning the entire database, the NUID physical record was developed. This physical record maps in-use NUID's to student/faculty identification numbers. Again, in one probe the UDB system can determine if a particular NUID is in use or not.

Design Features

There are design features of the UDB system that reflect the use of a tools based Unix approach. Among the more important topics, the following will be covered: universal computing identifiers, initial password generation, access control within UDB, shell strings, and client programs.

Universal Computing Identifier

A major part of the emphasis of the original UDB was devoted to the generation of "meaningful" user ids. Previously, all user accounts were generated in the form of "<prefix><count>" where each user shared a "<prefix>" with all of their classmates and had a unique "<count>". For example, a file structures class with three students would have their user logon account names generated as "fs1", "fs2", and "fs3". This further complicated the administration job by hiding who the person was behind a "fs1" account (as the faculty member assigned the accounts to the students). In addition, some accounts were duplicated, in that one student would be enrolled in several different classes, each with a different computer account for those class' programming on the same machine.

Therefore, more meaningful user names were desired. After a few experiments with the enrollment data, we developed the following algorithm. It is extensible, but has served our needs for the last two years.

A. First, all illegal characters are removed from the full name. These include such things as hyphenation and punctuation. Then the name is converted to lower case.

B. If a person's last name is greater than seven (7) characters and their first name is greater than three (3) characters and less than or equal to seven (7) characters then their first name is designated as "word1" and their last name is designated as "word2". Otherwise, their last name is designated as "word1" and their first name is designated as "word2".

C. Following is the list of checks for uniqueness. When one of these data base probes meets with failure (the UCI is not found in the database), that UCI is assigned to this user. The first one to be chosen ends the algorithm.
   1. The first seven characters of "word1".
   2. The first six characters of "word1" followed by the first character of "word2".
   3. The first five characters of "word1", the first character of "word2" and the first character of this user's middle name.
   4. The first seven characters of "word2".
   5. Their initials (i.e., first character of first, middle, and last names concatenated).

Initial Password Generation

In the past, accounts were typically generated without passwords or with a single initial password for the whole class. This posed several problems. Sometimes a malicious user would log onto as many of the accounts as possible and set a password on each account. This prevented the rightful user from logging in. Another case that has occurred is where a new graduate student has been given an account for the class he or she is teaching. These accounts were also generated without passwords. However, some of these new students didn't know that a password should be set, thus allowing anyone to log into their account where such things as grades, student id to student name mappings, etc. may be found.

After much thought, we came up with this solution. At account creation time, UDB generates a random password and associates it to the new account. The generation of the password proceeds as follows:

A. A random number is generated and modulo divided by the number of words in the word list. This word is retrieved and is called "word1".

B. A random number is generated and modulo divided by the number of separator characters. This character is retrieved and is called "word2".

C. Another word from the word list is chosen, as in "A" above, and is called "word3".

D. The password is generated by concatenating "word1", "word2", and "word3".

This word is then passed to the DES encryption algorithm that comes with UNIX to generate the encrypted password, and this is also associated with the account. It should be noted that on export versions of Unix, this will not work, due to the non-DES algorithm being installed in the libraries.

Both the plain text version of the password and the encrypted version are saved in the database. Only the plain text version is truly needed; however, the regeneration of the encrypted version is a very time consuming process (using the DES algorithm). It is for this reason that we generate the encrypted version only once and then save it for all future uses.

Users are encouraged to change their password as soon as possible, so that it no longer matches the UDB database. Their current actual password is *not* stored, only an initial value for the creation of a new account.

UDB Access Control

There is a layered approach to access control within the UDB system programs. First and foremost is user authorization. If the user running this image is explicitly allowed the ''udb_priv'' right, the user authorization code allows the function of the program to continue. If the user does not possess the ''udb_priv'' right, the typical action is to exit with a non-zero exit code.

After a user has been authorized, two lock files are used to attempt to maintain the database integrity from errors due to race conditions from multiple simultaneous UDB users. These two locks are maintained in the ''LIB'' directory, and are called ''LCK.PROC'' and ''LCK.DBM'' for the process lock and DBM lock, respectfully. Whenever a process (especially a shell script) starts to access the UDB, the ''LCK.PROC'' lock is established to maintain a shell level lock for the duration of the shell script. Then, each DBM accessing executable maintains and removes the ''LCK.DBM' while it is accessing the DBM. The typical reaction to the presence of a lock file where one should not exist is to simply wait. The lock files are of the type of lock files used by ''uucp(1)'' and others.

Shell Strings

To their credit, the Bourne shell and awk play a big part in UDB. However, to take advantage of the DBM database, a clean method of getting the data from the DBM to the shell and vice versa was needed. The Bourne shell already had environment variables of the form ''name=value''. It also has a method of evaluating a string of ''name=value'' pairs where each pair is separated by a semi-colon (i.e., ''eval''). Hence, ''udbget'' extracts the data from the DBM into a string of ''name=value'' pairs with each of the requested fields semi-colon separated. Further, ''udbput'' parses the same type of string and inserts the updated data back into the DBM.

Client programs

There is a server/client pair of programs for each of several operating systems. The special features of each are covered below. All of the client programs must be run by the ''super-user'' of the destination operating system, whatever user that may be.

AT&T Unix Sys V

This system implements a form of archival service for removing users from the system.

Xenix 286/310

Handles the OpenNET file system while attempting to balance the number of accounts to all systems that are in the system.

VAX/VMS

Passes the plain text version of the password so that running AUTHORIZE is made easier.

IdentaCard

Although not a system that anyone can log into, the IdentaCard computer is the department's card key access system. It is included here to show that seriously non-standard systems can be supported in a clean and efficient manner.

The Future

As presented, the UDB system is conceptually a relational data base. However, it is implemented as a multi-key hashed database using ''DBM'' routines by Ken Thompson. The relations are maintained outside of the DBM with one file for each key (via ''udbsel'' and ''rlgen.sh'').

During the normal operation sequence, the relations are generated, then the other programs use the key files to index into the DBM (using ''udbget'') to retrieve the student-key relation records. Each time a user's record is altered, the entire student-key relations must be regenerated. Having the records stored in a relational database would eliminate the need to regenerate the relationships each time the database is changed.

Roland Stolfa is a Software Specialist on the staff of the Computer Science Department of Oklahoma State University. His interests include network simulation, robotics, and system administration automation. Mr. Stolfa has worked for the university since 1986. Reach him via electronic mail at rjs@a.cs.okstate.edu. His U.S. mail address is: Computer Science Department; 219 Mathematical Sciences Building; Oklahoma State University; Stillwater, OK 74078.

Mark Vasoll is a Senior Software Specialist on the staff of the Computer Science Department of Oklahoma State University. His interests include wide area networking and improving the software facilities available at OSU to teach networking concepts. Mr. Vasoll has supervised the operation and expansion of the Computer Science research facility at OSU since 1985. Reach him via electronic mail at vasoll@a.cs.okstate.edu. His U. S. mail address is: Computer Science Department; 219 Mathematical Sciences Building; Oklahoma State University; Stillwater, OK 74078.

LISA IV
October 17-19, 1990
Colo. Springs, Colorado

Michael Urban – The RAND Corporation

GAUD: RAND's Group and User Database

ABSTRACT

GAUD mitigates the problems of managing large user communities on hetereogeneous networks. This paper describes GAUD, its database, and database operations. It discusses some of GAUD's functionality in addition to sharing some experiences using it.

Introduction

GAUD[1] is a system for the maintenance of user and group information on the RAND Computer Information Systems (CIS) network of UNIX systems. This network comprises about fifteen SUN-3 and SUN-4 NFS servers, about seventy-five diskless and diskful client workstations (including SUN-2, SUN-3, SUN-4, and SPARC workstations) and an aging (non-NFS) VAX or two. Some file servers are in remote locations, such as Washington, D.C., which constitute separate NIS and NFS domains. There is also a similarly-sized classified network that is completely separate from the unclassified network.

Some of the NFS server machines are also used as time-sharing machines via Ann Arbor terminals and serial lines. Such time-sharing users are charged for connect and CPU time according to the project they are working on in a particular session; all users are charged for disk usage on the NFS servers. To implement chargeback, the UNIX Group ID has been drafted into duty as an account number; this turns out to correspond well with its protection function in the RAND network. The *login* program has been modified to prompt the user for the account to use in this session (with a default account based on the Group ID found in the password file). The use of account numbers for groups makes RAND's group file unusually large, with over a thousand lines. The `/etc/group` file is not shared via NIS. Searching a group NIS database. for all the groups associated with a particular user ID requires a linear search (and consequent NIS fetch) of the entire database, since the database cannot be keyed on User ID, unlike the password database. With such a large group database, it was found that such searches are too slow, and can even time out when the server is busy. Instead, `/etc/group` is distributed to all clients nightly via *rdist*.

User home directories may live on any of the NFS file servers (or on non-NFS machines). NFS cross-mountings may occur even across NIS boundaries, and user directories may be moved around at any time. For these reasons, we have found it convenient to maintain consistent group and user information among all our machines, regardless of NFS or NIS affiliations.

Since 1981, the password and group files were maintained by a program called *adduser*. Originally, *adduser* was created to simplify the task of adding and modifying entries in the password and group files of a PDP-11 UNIX system, and was only a semi-automatic procedure; the users (the RAND CIS Business Office) hand-checked the results before manually installing the new password and group files. Over the next several years, diverse hands increased its user friendliness, scope, and complexity. In 1987, for example, when *adduser* was ported to SUN systems, a new "shadow group" file was added to the system to maintain the correspondence between group IDs and RAND account numbers, which had previously been accomplished by a hack to the format of the VAX group file.

Because *adduser* grew by accretion, it became harder and harder to repair bugs and adapt it to new demands. Help requests from the Business Office became ever more frequent. It was clear that it was time to replace *adduser* with a system built from scratch.

System Architecture

There are three main components to the GAUD system. The first is the GAUD database. This is a centralized database containing all the data associated with each user (generally identical with the information contained in `/etc/passwd`) and groups (including the mapping between formal RAND account number and UNIX Group ID). There is only one copy of this database, on a single machine. A GAUD dæmon maintains the integrity of this database and services requests from other (possibly remote) processes one at a time. Some potential problems of simultaneous update are therefore avoided without resorting to complicated locking mechanisms.

The second component is a collection of library calls to update the GAUD database. These calls are based on the Remote Procedure Call (RPC) protocol

[1]pronounced as in Latin, to rhyme with *crowd*.

and generally perform single simple tasks like *get-user* and *change-home*. The calls that modify the database use RPC's authentication mechanism[2] to restrict access to `root`. The GAUD library calls are used by a collection of eighteen client programs that the Business Office uses as GAUD commands. These clients provide robust and user-friendly interactive front-ends for the built-in GAUD library routines.

The third component is a special client that runs nightly on each machine that maintains a password file – masters for each NIS domain, and non-NFS hosts. This program, known as *gaudy-night*, rebuilds the `/etc/passwd` (or `/etc/passwd.yp`) and `/etc/group` files anew from the information in the GAUD database.

The GAUD Database

The heart of the GAUD system is the GAUD Database, comprising all the information pertaining to groups, accounts, and users. The database is currently implemented with the UNIX *ndbm*(3) package, and in fact there are two separate databases: the group database contains the mapping between RAND account names, group names, and group IDs; and the password database contains information on each user such as password, home directories, and so on.

Each record in the group database contains these fields:

Account Name The account name associated with the group. The group database is organized so that records are indexed and retrieved on the basis of this field.

Group Name The UNIX name for this group. Typically (but not always), this will simply be "`act`" followed immediately by the account name.

Group ID The numeric UNIX group ID for this group.

Description Arbitrary information associated with the group. The GAUD user may place any textual commentary into this field.

Hosts (*not used yet*) If this group is only to be used on particular machines, those machine names will be listed here.

The records in the password database contain the following fields:

User Name The name by which UNIX and GAUD refer to this user; the user's login name. The password database is organized so that records are indexed and retrieved on the basis of this field.

User ID UNIX's numeric identification for this user.

Password The encrypted version of the user's current password.

Person Number The RAND person number of this user. For some accounts (such as the UNIX `root` account), there is no single human individual involved; the person number `u0000` should be used in these cases.

Description Text describing who this person is; usually their full name.

Shell The full name of the program that the user will run when signing on. Normal users will use `/bin/csh` for their shell.

Homes The user's home directories. Since the user's home directory is different in different NIS domains (or machines that do not use NIS), this field contains entries of the form

dom_1:dir_1, ... domain_n$:$dir_n$

Users with no home directory in a particular domain will have no account on those machines, i.e., there will be no password entry on those machines for that user. Note that there is no extra space between the names and the commas and colons that punctuate the list.

Groups The account names for the groups of which the user is a member. These are separated by commas (with no extra space).

The GAUD dæmon updates these fields in response to a series of RPC-based calls, such as *change_home* (to change a user's home directory for a particular domain), *get_group* (to fetch a group database record), and *add_group_user* (to add a group to a user's list of groups). Some of these calls (like *change_home*) are technically redundant, since they could be accomplished by *get_user*, record modification, and *new_user* (to replace the record). However, building them into the dæmon simplifies client programming, improves the reliability of field values, and reduces the likelihood of errors caused by two clients attempting to modify a single record.

Incidentally, the GAUD dæmon itself uses a 'local database' version of the same assortment of calls to accomplish its tasks. The use of RPC to build identically-called 'local' and 'remote' libraries of GAUD calls proved to be quite useful during the development and debugging of the whole system.

The *passwd* program has been modified on the CIS network to notify the GAUD dæmon when a user changes his or her password, so that the database can be synchronized with reality. If the GAUD dæmon is unavailable for some reason, the request is queued and retried at a later time.

When a user's `groups` field is empty, and a new group is added for the first time, the reason is normally that a user's funding has been suddenly re-established. GAUD therefore immediately notifies RAND's own password dæmon (local software that implements such functions as password aging) to

[2]RPC has, admittedly, rather weak protection features. Our network is isolated from the Internet and other external networks, so RPC's protection is adequate for our purposes.

update all password files with the new information so that the user can sign on immediately. If this update fails, the user must wait until the next day to sign on. Other GAUD transactions do not take effect in the UNIX password and group files until these files are rebuilt when *gaudy-night* runs on password-maintaining hosts on the following night.

GAUD Commands

The GAUD user community is a small group of Business Office personnel who possess little technical training or confidence in UNIX procedures. The GAUD library was developed before the user interface was designed, on the basis of their experiences and analysis of their normal transaction mix. The library routines were tested with simple driver programs that each did little more than call one GAUD library routine based on command arguments. After a little time using the driver programs, it became apparent that these could become a perfectly reasonable working set of commands, and that such a set would be more easily maintained than some large, command-driven (or *curses*-driven) do-it-all program. Further, RAND's own familiar MH mail handling system, which is similarly constructed of small special-purpose programs, showed how many UNIX facilities (shell programming, printing, mailing of results) are provided 'for free' in such an environment.

The user programs were designed for the naïve GAUD users. Error messages are complete sentences and the command names are complete words (linked by hyphens rather than the more 'jargonish' underscore). Command syntax errors produce an informative command syntax description, and, since almost all commands require at least one argument, the users can get the syntax of almost any GAUD command simply by typing its name. The commands that require a lot of input (such as *new-user*) provide a lot of prompting and confirmation before a record is actually written. Other commands print out the newly modified versions of user or group records as they are modified, in a verbose, readable form (a 'compact' option is available for use in pipelines and shell scripts). Finally, all of the commands are in their own separate directory (with a group protection that allows access only to the Business Office and system administrators); this means that the users can always get a 'menu' of GAUD commands by using the standard UNIX *ls*(1) command.

It was clear that, even with robust and user-friendly commands, the 'naked' UNIX Shell environment might be considered intimidating to some of the users of this new system. For this reason, the GAUD user manual was designed to be more than a terse reference volume; it was written in informal language (and printed in large type) to enhance its accessibility, with plenty of background information and annotated examples of use. The importance of a 'reader-friendly' user manual should not be underestimated.

There are eighteen GAUD commands. These can be divided into the following groups:

- Group record manipulation: *new-group*; *destroy-group*; *get-group*
- User record manipulation: *new-user*; *destroy-user-record*; *change-user*; *get-user*
- User group list modification: *add-group-users*; *add-user-groups*; *change-default-group*; *delete-group-all*; *delete-group-users*; *delete-user-groups*
- Home directory modification: *change-home*; *delete-home*; *undelete-home*
- Record display: *print-rec*; *print-group-rec*

Group Records

The commands in this section are the only ones that deal specifically with the group database.

new-group

This command adds a new record to the group database. It requires an account name as a command line argument. All the information may be supplied on the command line; usually only the new account name needs to be specified, with GAUD automatically assigning an unused UNIX group ID and group name based on the account name.

The new group record is displayed when the operation is complete.

destroy-group

This command removes a record from the group database. The command-line argument provides the account number for the group to be destroyed. The GAUD user is given the option of removing the group from the group lists of all users in the password database before removing the group record.

get-group

This program retrieves records from the group database. The user may supply more than one account number on the command line. If the user supplies *no* account numbers, *all* of the records in the database will be retrieved and displayed (in no particular order). Groups may be specified either by RAND account name or UNIX group name.

User Records

The commands in this section deal with complete records in the password database.

new-user

This command adds a new record to the password database. It requires a user name as a command line argument; if the name is already in the database, an error is printed. In general, this command is used interactively as the user supplies all the fields for the record in response to queries from the program. However, there are an assortment of command-line flags that can be used to supply particular field values.

New-user will not update the database until the user confirms the record as correct, and the interrupt character can be used at any time to harmlessly abort without modifying the database (a reassuring message is printed in this case).

destroy-user-record

This program entirely removes a user's record from the database; the names of the users to remove are given as command-line arguments. For each user, their record is printed, and confirmation is requested (the operation may be aborted by hitting the interrupt character). If a user's entry has any home directories listed *or* is still in any groups, the record will not be removed.

change-user

This command allows the GAUD user to change various fields in a user's record; the user name must be supplied as an argument. It is basically the same program as *new-user*, with initialization of the 'working' record from the existing user record rather than defaults and command line arguments.

get-user

This command displays the password database records for each user named as a command-line argument. If **no** command arguments are given, *all* of the records in the database will be printed (in no particular order).

User Group Lists

These commands are used to change the group list for a user, or several users.

add-group-users

This command adds a single group (whose account number is the command's first argument) to the group list of one or more users. The resulting user record is printed out for each user.

add-user-groups

This program is very similar to *add-group-users*, but its first argument is *one* user name, followed by one or more group names. Business Office practice requires both styles of update in about equal parts, so two commands were provided.

change-default-group

This command neither adds or removes groups from a user's group list; instead, it moves one of the groups in the list to the beginning of the list. The first group in the list is always a user's *default group*; this is the group whose ID will be put in a user's password file entry (if this group is removed from the user's list, the second group in the list becomes the user's default group). The first argument is the user name; the second is the group to move to the front.

delete-group-users

This command removes a single group from the group lists of one or more users. The first command argument is the account number of the group to be removed; the remaining arguments name users to remove from the group (i.e., from whose group lists the group is to be removed).

delete-user-groups

This command is very similar to *delete-group-users*, but its first argument is *one* user name, followed by one or more group names to remove.

delete-group-all

This command removes one or more groups from *all* users in the password database. As each group is removed, a confirmation is printed. Unfortunately, the removal of each group from all the user records is done as a single transaction by the GAUD dæmon, and so it is not possible to print a message for each user removed from each group.

Home Directories

The commands in this section modify the Home Directory field in a single user's password database record. These functions should always be used in preference to *change-user*.

change-home

This command changes the Home Directory field for a user; the user's login name is supplied as the command-line argument. If no other arguments are supplied, the user is prompted for the information in a manner similar to *change-user*. Complete change information may be provided on the command line, however.

delete-home

This command may be misleadingly named; it does not actually delete any directory from the disk. Instead, it marks a user's directory on a machine (or several machines) as *moribund* by replacing the colon (as in "`twain:/home/twain_a/luser`") by a "#". During the next nightly run, a GAUD program on that machine will actually delete the specific directory (first backing its contents up to an archive file) and remove the directory entry from the user record.

The user may use the `-f` (Force) flag to actually remove the home directory entry from the record.

undelete-home

This command does just the opposite of *delete-home*; it replaces a "#" separator with a colon, effectively un-marking a previously moribund directory.

Record Display

Most GAUD commands display user and group records in a standard readable format. Many have a `-c` option to produce single-line records that can be handled well by UNIX tools like *sort*(1) and *grep*(1).

Two commands convert these single-line records back to the usual readable format: the *print-rec* command formats user database records; *print-group-rec* formats group database records. They use no command line arguments.

Gaudy Night

Each night, the main password and group files on various machines are synchronized and re-created from the GAUD database. This is accomplished by a GAUD client program called *gaudy-night.* It runs separately on each machine on the network that maintains a password domain – NIS masters, or non-NIS hosts. *Gaudy-night* scans the entire GAUD group and password databases (via normal requests to the GAUD dæmon), and does the following:

- For each record in the group database, finds all the users in the password database that have that group in their group lists. This will enable it to properly build an `/etc/group` file.
- For each user in the password database, it checks that user's directory for the machine on which it is running. There are three cases:
  - If there is no home directory at all for this machine, then the user will not get an entry in this machines `/etc/passwd` file.
  - If the home directory is specified, but does not exist, a new directory is created (with the appropriate owner and group) and is initialized with standard 'dot files' (e.g., `.login`, `.cshrc`).
  - If the home directory is marked as 'moribund' (as a result of *delete-home*), the contents of that directory are preserved (using the standard UNIX utilities *compress* and *tar*) and the directory itself is destroyed. The GAUD database entry for that user is updated by removing that home entry for the user. If the user has no more home directories, and is a member of no groups, the user's GAUD password database record will then be wholly expunged.
- Writes a file formatted like `/etc/passwd` (but not sorted) for all the users that have home directories on this machine. If a user is not currently a member of any group (i.e., has no account to which usage may be charged), a special login shell and default group will be placed in the password file so that the user can be notified of this state of affairs when next attempting to log in.
- Writes a file formatted like `/etc/group` (but not sorted) for all the groups, complete with the names of all the users in that group.

Gaudy-night must, in turn, be run by some sort of script that performs system-dependent postprocessing. All systems, for example, will want to sort the `passwd` and `group` files; what gets done with the result will depend on such things as whether the system runs NIS, whether a group-name/account-name mapping database must be built, and so on. This script may also handle such things as *gaudy-night* failure (due to dæmon unavailability) by retrying or notifying the operations staff.

A second, and unrelated, activity may take place at night on the various 'home' machines: finding their emptiest home directory system. Periodically (but not necessarily nightly), a script[3] finds the emptiest file system that might be used for user home directories, i.e., the system with the largest pool of free space. The name of this file system is copied to a centrally-located file whose name is the same as the machine name. This information is used by *new-user* and *change-home* when adding a user to a new machine.

Note that this activity is entirely independent of the GAUD group and user databases, and exists entirely as an advisory convenience for the GAUD user when adding new users to a machine. In other words, if this system fails (e.g., if one or more systems' 'emptiest file system' entries are not up to date), the problem is not critical.

Experiences

GAUD was first installed on a trial basis at the end of April, 1990, on a non-NFS VAX. This represented, in effect, a one-machine 'domain' that could be used as a GAUD testbed with limited effect in case of problems. During the next three months, GAUD was used daily, alongside of *adduser*, which was still used for password and group transactions for the rest of the CIS network. At the time of this writing, the system is within days of being put into operation for the entire unclassified RAND CIS network.

The Business Office quickly became acclimated to the GAUD environment for day-to-day operations. The users' normal practice has been to *cd* to the GAUD program directory, rather than adding it to their search path. They often use the *ls* command to remind them of the GAUD command names, and frequently force a 'syntax' message, notwithstanding the fact that a one-page GAUD reference sheet is available.

Reaction has been universally favorable, largely because *adduser* is often such a frustrating experience. Since corresponding GAUD and *adduser* transactions are both performed in a single day's login session, comparisons of the two systems were inevitable. Users have commented several times about the

[3]based on the general utility program *perl*

greater speed with which they can use GAUD, as well as its more instant feedback. Like many such programs, *adduser* collects a whole batch of changes to the password and group files, and applies all of them when the user exits. By comparison, each GAUD transaction is quick – the user can select any reasonably unloaded system on which to use GAUD – and the user knows the operation has been completed when the next UNIX prompt appears.

The users also feel more 'in control' of the GAUD database than they felt with *adduser*. The GAUD programs allow them to modify any field of the database specifically and individually. If they have made a mistake, they can correct it easily.

Future Work

The next step, of course, will be to establish GAUD on the entire CIS network, and stop all use of *adduser*. The more extensive use of GAUD may point up some hitherto unsuspected limitation or missing functionality. We expect, however, that the system has been designed to be modular and flexible enough to allow new database fields, library calls, and user programs to be added with comparatively little effort. When GAUD is firmly established on the unclassified network, it will be installed on the classified network; no problems are anticipated with such a transfer.

Because of the client/server architecture, we may later write a GAUD client program for the X11 Window System to give the users a more 'friendly' front-end. At present, the Business Office does not use workstations, so this is not likely to be contemplated for some time to come. Clients on other systems that are on the same physical network, such as IBM or Apple computers, might also be possible (if such machines become important in the Business Office's operations); we have not yet investigated the feasibility of RPC programming on such platforms, nor the security implications of such a client.

The GAUD system is tailored for RAND's particular needs. It is freely available on request with no promise of suitability. Contact the author for further information.

Reach the author at The RAND Corporation; 1700 Main Street; Santa Monica, CA 90406-2138 or at urban@rand.ORG electronically .

LISA IV
October 17-19, 1990
Colo. Springs, Colorado

newu: Multi–host User Setup

Stephen P. Schaefer – MCNC
Satyanarayana R. Vemulakonda – formerly of Duke University Computer Science Dept.

ABSTRACT

This paper describes a program, mostly composed of shell scripts, that installs user accounts on hosts across the network. The automation of this administrative task was, for the most part, uncomplicated. We discuss reasons for our decisions about account construction. The program improves speed, accuracy and task completion over our previous new account procedures.

Introduction

Adding a user to a machine traditionally involves explicitly creating an entry for the new user in `/etc/passwd`. And creating the user's home directory. And putting in several default dot files. And adding the user to groups. And adding the user to the list of aliases. And setting file system quotas. And one or two site-specific requirements such as insertions into accounting databases or database access lists.

The process is cumbersome and can be error prone. Several vendors provide administrative tools for the task, but, needless to say, they differ. If the new user has to be added to a group of machines the low–level process becomes repetitive and all the more tedious and error prone; or, worse, you deal with several different user addition programs which perform different subsets of the entire task, and then attempt to remember and remedy what each program couldn't do, or did inappropriately. Yellow pages is commonly available, and provides a subset of these features, but we don't run it at MCNC because it has historically been a security problem. At MCNC we have written a screen based utility that automates the process of adding new users to one or more networked machines.

MCNC is a consortium involved in cooperative research between academia and industry. We have about 45 computers running various flavors of UNIX, and a new user is typically added to more than one machine. *newu* is a shell script which displays a form on the screen: the administrator enters the required information once and the task of adding the user to various machines is handled by the script. At MCNC, every user is required to have an account with a low quota on speedy.mcnc.org, which is mainly used for reading mail and reading news. That account is used as a template, and comes in handy in other ways explained later. Users are added to the other machines as need be.

The *newu* Script and Its Invocation

The *newu* script consists of two parts: a user interface on the machine from which the accounts are being added (speedy) and *rnewu* on the machines to which the new user is being added.

The administrator invokes *newu* on the host machine, which uses shellforms (*sf*) to present and process two text screen forms, suitable for any terminal with a reasonably capable `/etc/termcap` entry. The shellforms program was written by Paul Lew of General Systems Group, Inc., Salem NH, who posted it on Usenet. The *sf* program takes a file as a form template and displays it on the terminal to perform basic form editing function. It also contains hooks to do more extensive field processing, which we use to good advantage: we changed a couple `#ifdefs` in the original code, and added an object module to do queries on `/etc/passwd` and other files. The *sf* program generates a shell script consisting of variable assignments, and our script does the remaining work. A verbatim illustration of the particular screens we use is given at the end of this paper. The next few paragraphs explain the fields found there.

Idiosyncratic processing occurs on the login name, user category, uid field, and password. If the administrator enters an existing login name or uid, we scavenge the contents of the remaining fields from the appropriate system files. For new accounts, the user category specifies a range of uid numbers from which to obtain an unused number. Specification of the user id ranges occurs in a text file, the format of which was determined by our previous new user program, *addu*. When we built *addu*, the correspondence between user category and user id range was thought to be important, but noone here now can remember why, nor identify any software that depends on those constraints. We kept the feature because it was easy to do, and, who knows, we might someday remember why we did it.

If the user exists, the password field appears with asterisks, and an internal variable holds the encrypted passwd; otherwise, the password field gets the cleartext, and the variable for the encrypted version is set to the null string, with encryption handled later in the script. As far as we can tell, the greatest risk of exposure of the cleartext password comes in the the shell script generated by *sf*, stored in a temporary file; that file, however, is owned by root and has mode 700 from its creation. At no time does the password appear as a process argument.

We have a convention for naming login directories: `/mcnc`/*user_category*/*login*. In many cases (usually to balance file system use), the home directory resides on a file system other than `/mcnc`, in which case the "conventional" home directory is actually a symbolic link to the "real" home directory. The second screen allows the administrator to install the user on a file system other than `/mcnc`. The "server" field is present in anticipation of automatically enabling access to our FREEDOMNET[1] distributed computing environment, but is not yet in usé; FREEDOMNET administration is performed outside this program.

Note that each computer has a separate home directory for each user; we do not follow the common practice of having a single system–wide home directory for each user which is then made available via NFS. Although subject to occasional review, we came to this decision for a number of reasons. First, we don't run NFS. When NFS originally became available, we concluded that its security problems were unacceptable, and that FREEDOMNET was a superior alternative. We *like* UNIX file system semantics, which FREEDOMNET preserves. Unlike NFS, FREEDOMNET also offers transparent access to remote devices, which we use to tremendous advantage in our backup system.

Another reason is that we run a significant variety of architectures, our sophisticated users have executables in their home directories, and those executables need to be different on different architectures. We currently run VAXes, DECstations, Sun 3's, Sun 4's, and a Convex C2. FREEDOMNET mitigates some of this problem: it examines object code and then transparently runs the program on the appropriate architecture, suffering only some network delays.

Using symbolic links, it would thus be easy enough to implement a single home directory for each user, but there are other reasons for separate home directories: no diskless workstations. Before diskless workstations were commonly available, before the number of hosts we were running grew, and before the hosts per architecture ratio was as high, our "power users" (mostly chip designers and software developers) were uninterested in common home directories, and, furthermore, the cost in speed of going across the network together with the cost in reliability of requiring two machines to be up instead of just one, was too high, regardless of the brand of network file system in use. Most of those outside of the power user group had accounts on only two or three hosts, only one of which they used daily.

When workstations first became available, we looked at these issues again, and came to the same decision. Now that we've had much more experience with workstations, we find them being used primarily as X terminals. Had X terminals been available when we were buying workstations, we'd like to believe we're sufficiently prescient to have bought them instead. For now, the workstations are serving as quite capable X terminals, and will continue to do so until (a) we don't have to install our own security fixes into X and (b) the dollar cost to maintain the workstation begins to approximate the purchase price of the X terminal. So, finally, we are currently contemplating replacing some workstation home directories on disk with access to home directories on another machine — whether via NFS or FREEDOMNET is yet to be decided. Our decisions are not set in stone; changes to *newu* will be part of the cost of changing our mind, but not a large cost.

Once the administrator fills the fields, *sf* stores the information in shell variables. *newu* then runs *rnewu* using a root *rsh*(1) to the affected machines, passing the appropriate values. (In this paper, *rsh* refers exclusively to the Berkeley remote shell command.) *rnewu* reads the variables. Until recently, root *rsh*'s were acquired using an MCNC extension to *rsh*[2]. Our version of *rsh* prompts the user for a password if access would otherwise be denied, and proceeds if the password is correct. The administrator types the root password for the remote machine, thus satisfying sufficient authentication to add new users. Currently, however, we have installed MIT's Kerberos[3], and even this nuisance (and vulnerability) has ceased.

The script massages the information into the form needed to add a new user to `/etc/passwd`. *pass*, a modified version of *passwd*(1), encrypts the user's password from stdin to stdout, to avoid *ps*(1) eavesdropping. Since the account isn't in `/etc/passwd` yet, we also put other components

[1]Bob Warren, Tom Truscott, Kent Moat, and Mike Mitchell, "Distributed Computing using RTI's FREEDOMNET in a Heterogenous UNIX Environment." *Proceedings of the UNIFORUM Conference*, pp. 115-126 (1987).

[2]Helen E. Harrison and Tim Seaver. "Enhancements to 4.3 Berkeley Network Commands." *Large Installation Systems Administration III Workshop* [September, 1989].

[3]S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. "Kerberos Authentication and Authorization System." *Project Athena Technical Plan*, section E.2.1

of the passwd entry on stdin to allow *pass* to reject inappropriate passwords. A loop over each machine entered in the second screen sends appropriate information to the remote script.

The remote script is fairly straightforward: first it reads the expected variables from stdin. It checks whether the login name is already present, exiting with failure if so. It constructs an entry for `/etc/passwd`. It loops until the file `/etc/ptmp` is gone, and builds a new `/etc/passwd` in `/etc/ptmp`. It checks for the BSD4.3 utility mkpasswd and runs it if available, otherwise, it simply moves `/etc/ptmp` to `/etc/passwd`. It creates the user's home directory according to the convention explained above. On machines which have quotas, it invokes *edquota*(8) with an *ed*(1) script on the stdin; so far we've been lucky that only one of our vendors of machines with quotas has tampered with Berkeley's format.

Once the home directory and quotas exist, it installs our default dot files, which include: `.cshrc`, `.profile`, `.login`, `.kshrc` and `.forward`. At MCNC we follow the mail convention that a user reads mail on a home machine. We direct the user's mail from all accounts on all machines except the home to speedy. From speedy the mail is forwarded to the user's home machine account. *rnewu* creates all the necessary `.forward` files to implement this scheme. Of course, the user is free to arbitrarily change the `.forwards`.

rnewu's last chore is to edit `/etc/group` to reflect any groups to which the user belongs beyond the one in `/etc/passwd`. We use a *sed*(1) script to create `/tmp/group` and then replace `/etc/group` with the result.

Our mail system sends mail addressed to unrecognized users to speedy (where everyone has an account); consequently, you only need to know the correct login name within MCNC, and, further, mail aliases need only be maintained on speedy. It follows that *newu* itself changes `/usr/lib/aliases`, which it does similarly to *rnewu*'s editing of `/etc/group`, after which it runs *newaliases*(1). Processing is then complete.

Experiences

On a Microvax II running 4.3BSD with a load average of about two, *newu* takes between 5 and 12 seconds to start up. Given an existing user, it takes another 7 to 12 seconds to garner the information it needs from the half dozen or so files it looks at. For a completely new user, it takes about 4 seconds to generate an unused user id number and home directory. The time it takes to install a user on a machine depends on that machine's speed: on our hosts it can take anywhere from a few minutes for a heavily loaded VAX 11/750 to a couple seconds for a Convex C2. Because the program replaced a slower one that required the administrator to first log in to each machine on which an account was to be installed, we haven't yet had any complaints about speed. Kerberos authentication has further decreased the amount of time required for the program's operation.

If it were important to achieve a further speedup, we would first look at collecting information on multiple accounts, and creating those accounts in one *rsh* session to a given machine. We would also look at creating accounts on different machines in parallel, which could probably still be done from the shell script. At this point I am skeptical that rewriting the shell portions in perl or C would significantly increase speed (although *sf* can produce variable assignments in perl syntax).

There were a couple a problems in the scripts during the first couple weeks of use, due to programmer mistakes, but since then none. We have since added a facility to maintain a text database used in resource accounting which is present on some of our machines, using the same techniques we use to maintain `/etc/group`.

Consistency between utilities has not been a problem: on the one hand, as a BSD 4.3 source licensee, we are fully capable of replacing a utility that violates our notion of orthodoxy; we also have source code for any utility used here which was not provided with BSD 4.3, most notably *sf* and *ksh*(1). On the other hand, most vendors wisely avoid tampering with the fundamental utilities our scripts make use of: *sed*, *grep*(1), *ed*, *tr*(1), etc. The script has had to make allowances for differing quota formats.

Availability

Our modified version of shellforms and our scripts will be available for anonymous ftp from mcnc.mcnc.org. There should be no more than a few dependencies on *ksh*. Our home directory convention is unlikely to fit your environment, and may thus need modification. Root will be running this software: take appropriate precautions.

Conclusions

newu has been in use for more than four months, and has been heartily welcomed by the staff responsible for adding accounts. It makes their task of adding new users to various machines less tedious, less time consuming, and less error prone.

Screen 1

```
                      MCNC New User Account

 Login Name: sps______   User Category: mcnc____   Uid: 4121_
 User's Full Name:    Stephen Schaefer_______________________________
 Institution: MCNC                                Office Number: 217_
 Password: ********                               Phone: 2481417___
 Home Machine: pepe.mcnc.org______________________ Home#: 9426536___
 Home Directory: /mcnc/mcnc/sps______________________________________
 Shell (empty means /bin/sh): /usr/local/std/bin/ksh_____________

 Group Privileges
 mcnc_______     rootgrp_______     pa__________      unix-src_______
 Mailing Lists
 postmaster_________     sysadm__________
 root_______________     ris_____________
=====================================================================
                  Type Ctl-\ for the next form.
=====================================================================

[RETURN] forward  [CTRL-T] back  [CTRL-L] redraw  (arrows)
```

Screen 2

```
                List of machines with this account

 Machine    Server    Real_Login_Directory  DiskBlockQuota:  DiskFileQuota:
                                            Hard     Soft     Hard    Soft
 _________  ________  /mcnc/mcnc/sps______  2000     1800     500_    450_
 _________  ________  /mcnc/mcnc/sps______  2000     1800     500_    450_
 _________  ________  /mcnc/mcnc/sps______  2000     1800     500_    450_
 _________  ________  /mcnc/mcnc/sps______  2000     1800     500_    450_
 _________  ________  /mcnc/mcnc/sps______  2000     1800     500_    450_
 _________  ________  /mcnc/mcnc/sps______  2000     1800     500_    450_
 _________  ________  /mcnc/mcnc/sps______  2000     1800     500_    450_
 _________  ________  /mcnc/mcnc/sps______  2000     1800     500_    450_
 _________  ________  /mcnc/mcnc/sps______  2000     1800     500_    450_
 _________  ________  /mcnc/mcnc/sps______  2000     1800     500_    450_
===========================================================================

[RETURN] forward  [CTRL-T] back  [CTRL-L] redraw  [CTRL-\] send  (arrows)
```

Stephen P. Schaefer has been a systems administrator at MCNC since 1987, and is contemporaneously pursuing a Masters Degree in Computer Science at the University of North Carolina at Chapel Hill. Before that he performed systems administration at Bowling Green State University in Ohio. He has a bachelors degree in Mathematical Sciences with a Computer Science Option from the University of North Carolina at Chapel Hill. His current address is: Stephen P. Schaefer; MCNC; P.O. Box 12889; Research Triangle Park, NC 27709. Reach him electronically at sps@mcnc.org.

Satya Vemulakonda was an undergraduate at North Carolina State University, and worked as an intern at MCNC for the summer of 1989. His last address was: Satyanarayana R. Vemulakonda; 2518-102 Avent Ferry Rd.; Raleigh, NC 27606.

LISA IV
October 17-19, 1990
Colo. Springs, Colorado

Uniqname Overview

William A. Doster, Yew-Hong Leong, and Steven J. Mattson – The University of Michigan

ABSTRACT

This paper describes Uniqname – a package that enables the coordination of both UID and login name allocations (and optionally Kerberos password management) by a decentralized collection of system administrators without interfering with their autonomy over other issues such as login password, machine access authorization, account creation/deletion, and home directory placement. The paper also includes the problems Uniqname addresses, the various design points worked out for it, the resulting overall package, the client-server transactions in brief, and three anticipated usage scenarios. Source code is available through AFS and anonymous FTP.

Introduction

This paper describes Uniqname – a package that enables the coordination of both UID and login name allocations (and optionally Kerberos password management) by a decentralized collection of system administrators without interfering with their autonomy over other issues such as login password, machine access authorization, account creation/deletion, and home directory placement. The paper also includes the problems Uniqname addresses, the various design points worked out for it, the resulting overall package, the client-server transactions in brief, and three anticipated usage scenarios. While designed to work in a university environment, the overall approach should be flexible enough for use in a variety of environments (and for a variety of operating systems).

Terminology

One of the things that became clear during the many discussions leading up to the final design of Uniqname was that as system administrators we often use several terms interchangeably even though, in an abstract sense, they do mean different things. For the purposes of the paper then, the following terms are defined to mean:

User – The person that a given login name and UID will be associated with.

Login Name – The name that the user uses to identify her/himself to the operating system at login time.

UID (Unix ID) – The number stored in the owner field of UFS file systems.

System Administrator – Anyone authorized to handle and be responsible for user name creation and deletion.

Account – In general, all things to everyone, but in this paper having an account means that the system administrator has taken whatever steps were necessary on the target machine to allow the user to log in under her/his login name.

Background

In order to understand our motivation and implementation, the following brief description of the state of UNIX account administration at the University of Michigan prior to the use of Uniqname is provided. You may find it helpful when considering whether Uniqname would be appropriate to your site.

In the past, as various colleges and departments within the university had need of workstation-class resources, each department purchased and administered each machine or set of machines independently. As workstations became an integral part of the curriculum in the College of Engineering, the Electrical Engineering and Computer Science (EECS) departmental computing organization and the Computer Aided Engineering Network (CAEN) for the entire Engineering College were formed. Because of the large overlap in users, these two organizations have shared a good portion of their accounting information for a few years.

As more and more colleges and departments have invested in workstation-based computing, maintaining compatibility for users with several accounts and at the same time security has become difficult if not impossible for individual administrators. Uniqname, developed jointly by CAEN and the university-wide Information Technology Division (ITD), is our solution to this problem, usable by the entire university community.

Why Change?

There are a variety of reasons why common name and UID spaces are desirable. In general though, they make movement and communication between autonomous systems easier and, at the same time, lay the foundation for future campus-wide services. The following sections detail some areas that system administrators and users are most likely to benefit from.

System Administrators

Having common name and UID spaces makes it easier for systems within the common space to exchange dump and tar files because the UIDs stored within them will reflect the same owner on all systems. Also, they make sharing file systems via NFS and/or AFS less confusing because programs that look up UID-to-name in `/etc/passwd` will report the correct name (such as `ls-l`). Uniqname also allows system administrators to offer users a computer program to help them choose their login name from among the remaining valid names across all systems (see "Name Choice: Admin vs. User").

Users

Users in general prefer to have the same login name on all the systems they use – fewer things to remember. Plus it's easier to look up and remember a friend or colleague's login name if s/he has the same one everywhere. Also, users benefit from shared filesystems in the same way that system administrators do, as well as benefiting from any system-wide services that system administrators are then able to provide.

Future Services

A common name space also facilitates the development of future services intended for everyone within it to use. As an example, one immediate fallout from the Uniqname project is a University-wide Kerberos Authentication database that network services can base authorization decisions on. Another example is the registration of all Uniqname entries in the umich.edu AFS protection database, which allows files in the umich.edu AFS cell to be permitted to these users and enables users to authenticate for such access.

Some installations may consider mail names and login names to be interchangeable, as we do. For such installations, Uniqname provides a name space upon which campus-wide mail services may be layered. At University of Michigan, we intend to use Uniqname with X.500 to provide mail service. This will facilitate sending mail to either login@umich.edu or first.last@umich.edu.

Design Points

This section deals with the various points that influenced Uniqname's overall design. For each of the points, it presents two possible approaches, outlines the one we chose, and tells why.

Identification: Heuristic vs. Unique Key

Our key goal was for each person on the campus to have one, and only one, campus-wide allocated name and UID. To achieve this, we needed to avoid both collisions (different people with the same login name or UID) and duplicates (same person with different login names or UIDs). We boiled the numerous ways of doing this down to two approaches: by heuristic and by unique key.

In the heuristic approach, the system administrator accesses (possibly via finger, or other similar means) some well-known set of machines for the desired login name, and perhaps the person's first and/or last name. If nothing comes back, it is probably OK to issue the login name to a new user. If something comes back, the system administrator needs to check to see if more than one person "owns" the login. If only one person uses it, then it should be used for that person's local account. If more than one person uses it, an alternate login will have to be selected. Some areas that will be checked by system administrators will be the school or department as well as phone numbers in ".plan" files. UIDs will be allocated from a range of UIDs assigned to that particular system administrator.

In the unique key approach, the system administrator requires that the person asking for the entry provide a unique key that only s/he would have. This number would then be indexed into a database of login names and UIDs to check whether that user already had an entry (in which case it should be used for the local account) or whether a new entry should be added and a newly allocated name and UID returned.

In the end, we decided on the unique-key approach even though what would be used as the unique key and whether the database server would allocate UIDs from system administrator ranges or from a common pool was still undecided.

UIDs: Ranges vs. Common Pool

Previously, UID allocation had been made manageable by handing out ranges of UIDs for each system administrator to allocate from along with some special ranges for system accounts and accounts local to the machine. Our original allocation looked like:

UID Range	Usage
0-99	System accounts (root, uucp, ...)
100-999	Accounts local to machine
1,000-11,999	CAEN
12,000-15,999	CAEN/ITD Joint Accounts
16,000-29,999	non-CAEN/EECS Accounts
30,000-32,765	EECS

The CAEN/ITD joint accounts were used to administer our first campus wide access lab, but with some difficulty.

This appeared to be a workable approach until we started asking each of the campus units outside of CAEN and EECS how many UIDs each would need out of the 16-30 range. Naturally, they all wanted to ensure room enough for future growth and so each asked for rather liberal amounts of UIDs. When added up, the total was more than twice the number of UIDs available[1].

An alternative approach was a UID-server (such as uniquid[2]) This would allow efficient use of the UID space, but would make system administrators depend on a central service. Once we decided to use a central server for name allocation, this became a moot point (later versions of Uniqname plan to use a distributed database to reduce single-point-of-failure concerns). Therefore, we decided to pool the normal UIDs and ended up with the following layout:

UID Range	Usage
0-99	System Accounts (root, uucp, ...)
100-999	Accounts local to machine
1,000-32,765	All User Accounts

Administration: Central vs. Distributed

Having decided on a central database server for both login name and UID allocation, we needed to decide whether administration of this database would be done by one central organization or by a distributed group of system administrators. We realized having to call or email a central organization every time a system administrator needed to create or delete an account was impractical (actually, given our degree of autonomy within each campus unit, it was never seriously considered). Thus, we agreed to store a list of system administrators who were authorized to make changes to the database. We further agreed that information maintained by one system administrator would not be modifiable by any other system administrator.

Authentication: Login name vs. Kerberos

Having agreed that operations on the database should only be possible by authorized system administrators, we were faced with the problem of authenticating who the person requesting the operation really was. Traditionally, login names/UIDs and host addresses are used as the basis for authentication decisions, but with NFS browser programs and rhost viruses, we were hoping for something a little more secure. Fortunately, the group charged with implementing Uniqname had experience with the popular Kerberos Authentication package, which was deemed good enough for our needs. For each administrative group, a Kerberos principal is created that is authorized to administer that group's login names/UIDs, and the password for that principal is given to that group's system administrator(s).

Name Choice: Sys-admin vs. User

One of the issues that all system administrators face, in one form or another, is that of choosing a login name for their new users. Sometimes there is a company-wide policy (''first name plus first two letters of last name'' or simply ''initials plus number to make unique''). Sometimes the system administrator makes one up rather arbitrarily. Many times the name arrived at isn't quite what the user wanted.

It would be nice (should system administrators for a given campus unit allow it) if users could choose their login names themselves (on a ''first asker wins'' basis). To support this, we broke login name allocation into two phases. First, a computer-generated name is allocated in the form <initials><uid>. Second, either the system administrator who originally allocated the name or the user her/himself is allowed to choose a new name according to the following restrictions:

- The login name must not already be allocated.
- The login name must be three to eight characters in length.
- The login name must be composed only of lower case letters.
- The current login name must have numerics in it.

The first restriction is axiomatic to the one-person, one-name goal.

The second two restrictions help ensure that the login names people choose will be acceptable to as many of the operating systems (UNIX, CMS, etc.) on campus as possible while still allowing some room for personalization.

The last restriction makes it very difficult to change the login name more than once. This helps avoid the confusion of frequently changed login names, and it limits the abuse of name changes. One common abuse has been with users changing their names to something offensive (or misleading), sending mail to someone, and then changing their names back to their original settings[3]. With Uniqname people are only given one campus-wide name and are only allowed to personalize it once. This effectively discourages them from choosing something frivolous (because they'll be forced to be known by it for the duration of their stay at the University except in extenuating circumstances).

Duration: Account vs. Lifetime recording

With most of the allocation problems now worked out, we turned to the final problem – de-allocation of login names and UIDs. The basic question to answer was how would the database

[1]Some UNIX OS's are limited to 15-bit UIDs.

[2]Uniquid – Presented during Tutorial Sections at LISA III Workshop. Contact Jeff Forys, forys@cs.utah.edu.

[3]There are of course other means of forging mail with current mailers, but that doesn't mean we want to make it any easier.

know when it was acceptable to remove the entry from the database. Initially, we thought that the database would simply keep a list of all machines that that login name had accounts on. We quickly discarded this idea once we considered the number of machines (well over 1,000), login names (currently around 10,000), and the frequency with which system administrators would need to make updates. Instead, we decided to simply have each group that wanted the name/UID to exist register the expected date on which they would stop caring whether the entry existed (presumably caring because that group had accounts for that login name on some of its machines). The section entitled "Expected Lifetimes" details how these lifetimes are used by the Uniqname package.

Umich.edu: UIDs & Kerberos Names

One of the side effects of pursuing a common name and UID space at the University of Michigan has been the creation of a campus-wide Kerberos Authentication Database (complete with Kerberos keys). The need for the authentication database arose mainly from the requirement for secure authentication of the system administrators and later users too (because the Change Name operation also needed to be authenticated). Even before the database was ready, developers across campus began coming up with potential uses for the campus-wide authentication server.

Another addition to Uniqname was the allocation of UIDs in the umich.edu cell, thereby allowing directories served by AFS file servers to be ACL'd to users authenticated under those UIDs. This allows all registered users to read and write files into a common file system.

Resulting Package

This section describes the various programs, concepts, and structures key to the final package.

Client, Server, Scanner

The package is composed of three programs – a client-server pair and a scanning program. All changes to the database are initiated by client transactions with the server. The scanning program is run periodically. It scans the database for unconfirmed name changes, entry transfers, and expired lifetimes and then sends email to the system administrator responsible for any that it finds.

Administrator List

A list of authorized Kerberos principals is kept on the server machine. A system administrator running the client program must be authenticated as one of those on this list to perform any of the transactions below (including queries). Most transactions will further require that an administrator be authenticated as the one with administrative control over the entry.

For each principal there is also an associated mailing address to which the scanning program sends mail.

Principal	Mail Address
ifs	sys-adm@ifs.umich.edu
caen	hobbes@caen.engin.umich.edu
lsa	emv@math.lsa.umich.edu
citi	dave@citi.umich.edu
rssun	wes@terminator.cc.umich.edu
rsapollo	brian_moore@um.cc.umich.edu
rscms	cel@terminator.cc.umich.edu

This approach seems to best preserve the autonomy of each campus unit.

Unique Keys

Unique keys are used by Uniqname both to ensure that system administrators don't identify one user as another and that they don't allocate multiple names to one user. Unfortunately, names and UIDs are not just used to identify users; they also serve to identify system and class accounts. Therefore, whatever key we decide on must be unique across the entire University – for entries tied to individuals as well as those tied to system and class (or group) accounts.

Class Keys

Our experience has been that entries not tied to a single individual are tied to either special system accounts (such as root) or to University-defined entities, such as departmental and class accounts.

Departmental and class accounts generally coincide with a system administrator's area of responsibility, and because of this, we can simply delegate the responsibility for generating unique keys to the various groups. We did this by treating unique keys as character strings and assigning each group a string prefix for the unique keys it needed to generate. Locally, the group administers a sequence number. Whenever the group needs to create a unique key, it simply concatenates its prefix and then bumps up the sequence number. As an example, if CAEN's prefix were "CAEN_", its first unique key would be "CAEN_1", followed by "CAEN_2." System administrators are encouraged to keep local records that will help them distinguish between various entries; however their failure to do so only causes confusion local to their department.

System Keys

System keys on the other hand require somewhat special handling. We expect both a limited number of these entries and very long lifetimes. Entries for these names are mainly intended to serve as placeholders (otherwise users might choose "root" as their name, which could cause at least some users confusion). Because the UIDs for such accounts differ from one flavor of UNIX to another, the UID recorded for these entries isn't expected to mean anything. Also, to mark them as special entries, the prefix for their unique keys is SYS_ (see "Database Queries" for special handling of these entries). Because one prefix is used for entries that are allocated University-wide, we expect that a fair amount of communication will go on between system administrators before creating such an entry. We have created a mailing list (uniqname-administrators@ifs.umich.edu) to aid this.

User Keys

Users, unlike class and system accounts, unfortunately move from department to department and even from college to college. We can not count on users to report if they have an extant name or what that name is; users forget and, in some cases, intentionally report that they don't already have a name (because some services allocate resources on a per-name basis). For these reasons, we decided to make use of the same number that other units on campus use to distinguish between accounts – the University ID. For most students, faculty, and staff, this number is simply their social security number plus a check digit. For foreign students, this number is their Foreign National number. A small number of students choose to be referred to by a University-assigned number instead of either of these. In all cases though, the number is unique across the University and therefore meets our needs.

Underlying Assumptions

It is important that system administrators understand the differences between the three classes of keys and the assumptions that underly them. Otherwise, they may mistakenly allocate prefixed unique keys in an inappropriate manner.

User keys are intended for people. Because a fair number of people move around it is important to make it easy for system administrators to handle such changes. Therefore, if system administrators can come up with a Univerity ID for a user rather than generating a prefixed one, they will be both reducing the number of prefixed entries (which they need to locally maintain) and making it easier to transfer administrative control of the entry later, should the need arise.

Class keys, on the other hand, are intended for entries that will always be administered by the system administrator that originally allocated them. We anticipate that class keys will be needed for departmentally-created entities. But since no central authority allocates unique keys for them, they also afford system administrators a relief valve of sorts; any entries that don't fit into the user key category can still be allocated by creating a prefixed unique key.

Unfortunately, this leaves the door open to abuse of class keys by allocating them to people who do have University IDs. This is an abuse because it makes it difficult for other system administrators to ensure that multiple entries don't exist through use of the University ID. Hopefully all system administrators will understand and choose to respect these guidelines. (Uniqname server logs are maintained, however, to aid detection of such abuse should it occur.)

System keys are only for entries relating to names reserved by various operating systems. These keys should never be used for any other purpose. The use of both the key and the name for such entries should be widely discussed by all system administrators before they are actually allocated.

Expected Lifetime

Each entry has a list of expected lifetimes: one for each administrator that cares about the existence of the entry. When the expected lifetime expires, the system administrator for that group receives an email message informing her/him that the group's expected lifetime for that login name has expired and that the system administrator should do one of two things: extend the lifetime or delete it altogether. If s/he extends it, no further messages are sent until the next time the lifetime expires. If s/he deletes the lifetime, no further messages are sent. If that lifetime was the entry's only remaining lifetime then the entire entry is removed from the database, freeing both the login name and UID for later reuse. If the system administrator neither extends nor deletes the lifetime, s/he will continue to receive periodic reminders to do so.

Database

The database logically only contains one type of entry. The physical layout of the database is version specific. The current version stores the database in a combination of text and ndbm files. The next version will use a replicated database. The final version is expected to use Ubik – a transaction-based database package that handles replication and network partitions. A sample entry in the resulting database would be logically structured as follows:

Field	Example
Unique Key	"3764832791"
Login Name	"billdo"
Prev Login	"bad3597"
UID	3597
Fullname	"Bill A. Doster"
Adm	"ifs"
Transferred	1
Group1	0, "ifs", 90/6/11, 99/12/31
Group2	1, "caen", 90/6/15, 91/5/12

This example represents an entry for the user known in real life as Bill A. Doster, whose University ID is 3764832791, login name is billdo, and UID is 3597. The entry is administered by the IFS group. Any transfer of administrative control has been acknowledged. Both the IFS and CAEN groups care about the existence of his entry: IFS until the turn of the century, and CAEN until somewhat after the end of Winter Term, 1991. Also, his previous login name was "bad3597"; and while CAEN has already confirmed the name change, IFS has yet to do so.

Transactions

All database accesses are done through the transactions that follow. There are some design points common to all transactions that modify the database:

[1] Each transaction is UDP-based.

[2] Each transaction is composed of a single query-response.

[3] Each transaction is mutually authenticated via Kerberos.

[4] Each transaction is fully encrypted.

[5] Each transaction is idempotent. Retransmitting doesn't hurt.

[6] All packet fields are in network order.

[7] Each transaction carries the number of the current attempt.

[8] All transactions are logged.

[9] All transactions have a common return format.

The first two design points reduce load on the server and also remove the possibility of a client program tying up the server. The third design point ensures to the server that the client is really authorized to perform the operation and ensures to the client that the server is really the Uniqname server and therefore able to perform the operation. The fourth design point prevents both modification (safe messages are currently broken under Kerberos) and interception of possibly sensitive information (such as University IDs). The fifth design point both thwarts replay attacks (nothing gained) and simplifies error recovery (simply retransmit until a reply is received). The sixth design point allows easy interoperation between different architecture types. The seventh design point allows detection of network problems or server overload. The eighth design point increases accountability, allows time-of-transaction queries, and aids detection of attacks on the server. The last design point simplifies packet marshalling/demarshalling on both the client and server ends. Response packets contain a return code indicating whether the transaction succeeded followed either by the resulting database entry or by ASCII error text explaining why the transaction failed.

As an aside, performing mutual authentication and the exchange of data normally requires at least two packet exchanges in Kerberos. Unfortunately this violates the single exchange design point. To get around this, we piggy-back the transaction data on the mutual authentication packets. This is acceptable because while both parties need to trust the other end, neither one of them needs to do so *before* sending data to the other party. The server receives all the information it needs in that single packet to both authenticate the client and perform the operation. The client doesn't mind throwing encrypted data to spoofers, it just needs to know that the reply it receives is authoritative.

Allocate Entry

Allocate Entry allows any system administrator to create the initial entry for a given unique key. It requires the unique key, the entry's full name, initial lifetime, and whether the current password should be (re)set. The adm field is set to the authenticated group that is performing the transaction. In addition, Allocate Entry allows a preferred name and UID to specified. If either the preferred name or UID are already in use, the entire transaction fails. Along with adding an entry to the Uniqname database, our version of Uniqname also creates corresponding entries in the Authentication and Protection databases used by umich.edu. Finally, in addition to the normal record returned, Allocate Entry also returns the password for the entry's Kerberos principal. Note that this password is not necessarily identical to that used to log in to any of the user's accounts, although it could be. It is, however, the password used to authenticate the user when s/he does a Change Name transaction and is also the password used to acquire tokens for the University-wide cell – umich.edu.

Set Lifetime

Set Lifetime allows a system administrator to either register her/his group's first lifetime for a given entry or to change the current one. It requires the entry's login name and (new) stop date. The "group" field of the lifetime is set to the authenticated group that is performing the transaction. This transaction only fails if the named entry doesn't exist.

Change Name

Change Name allows the owning system's administrator or the user her/himself to change the name of an entry. It requires the entry's login name and the desired new name. If the desired name is already in use, the entire transaction fails. If the desired name is invalid, the entire transaction fails.

Acknowledge Name Change

Acknowledge Name Change enables each group that has a lifetime registered for a given entry to confirm that it has done whatever is necessary on its computers to handle the entry's earlier name change. It requires the entry's (new) name. The group is implicitly the one the system administrator is authenticated as. The transaction fails only if the group doesn't have a lifetime registered for the given entry.

Delete Lifetime

Delete Lifetime allows the system administrator for a given group to remove that group's lifetime from a given entry. This indicates that the group no longer cares what happens to the entry and should only be done when the group no longer has accounts or any other system entities that depend on the existence of the entry's name and UID. The transaction requires the entry's name. The group is implicitly the one the system administrator is authenticated as.

Transfer Entry

Transfer Entry allows either the owning group's system administrator or the user her/himself to transfer administrative control of the given entry to some other group. The main effect this has is the transferral of the ability to set passwords. A side effect is that if the new group doesn't already have a lifetime registered for the entry, then a copy of the old group's lifetime information is registered for the new group. The transaction requires the entry's name. If any previous Transfer Entry remains unacknowledged, the transaction will fail.

Acknowledge Entry Transfer

Acknowledge Entry Transfer allows the new group's system administrator to acknowledge the given entry's transfer of administrative control to that group. Until the transfer is acknowledged, no further transfers can be performed. The transaction requires the entry's name. The transaction will fail if the group the presenting system administrator is authenticated to isn't receiving ownership of the entry.

Change Unique Key

Change Unique Key allows the owning system's administrator to change the unique key of an entry. It requires the entry's unique key and the desired new key. If the desired key is already in use, the entire transaction fails. If the desired key is invalid, the entire transaction fails.

Queries

Queries modify neither the database nor the log. They are handled differently from the above transactions. Queries are presented to the TCP port of Uniqname and cause a separate instance of the server to be forked off to handle the request since queries may be both time-consuming and generate large amounts of data. As before though, connections are both mutually authenticated and fully encrypted.

Each query is composed of a record specifying match conditions for each of the fields in either the database or log record. Each field also has an associated flag to mark whether the field can be ignored. The set resulting from the query shows all records that match all of the fields that are not to be ignored. To effect an `or'ing` of selection criteria, multiple queries would need to be made.

The reply to the query is a full copy of each of the matching records. No provision is made to retrieve only the fields of interest. Also, the Unique Key field is specially handled: it is blanked out unless the prefix is "SYS_" or the client is authenticated as the very restricted Uniqname principal.

Creation Approaches

The last section details three approaches to creating Uniqname entries. The approaches differ mainly by when the initial entry is created and whether the system administrator or user chooses the entry's name.

Batch Creations

The Batch Creation approach is most often used when large numbers of entries are created over a short period of time – usually by a script. The script is given a list of University IDs, from which it produces a list of login names, UIDs, and passwords (along with Full Names for ease of identification). All entries are administratively controlled by whatever group the script was authenticated as when it was run. All entries have the lifetime specified when the script was run. The login names and passwords are passed out to their owners who are then able to choose what their actual login name will be by doing a Change Name transaction themselves.

Administrator Determined

The Administrator Determined approach is most often used when the desired login name is already known or when absolute control over user names is desired. Since computer-generated names will not be used, the system administrator may need to go through several iterations by hand (or provide several alternatives to a script) to acquire an unused name. In this case, the intended user would have no *local* need for the returned password because her/his name is already final. Therefore the system administrator could put off handing out such passwords until the user needed it for umich.edu access. At that point, if the system administrator no longer had the password, s/he could do another Allocate Entry call with the Set Password field set and give the user the new password returned.

User Created

The User Created approach is most often used when the need for login names and UIDs doesn't exist until the user needs them (a counter-example would be if a professor wanted to permit files to a student after s/he had enrolled in the class but before s/he had registered for an account). The appeal of this approach is that a front-end program can be brought up on trusted machine authenticated as the group that will have administrative control over entries allocated. The program either prompts users for their University IDs or acquires them by having users run their University IDs through a card reader.

Next, it optionally checks if the given University ID is registered in the local (non-Uniqname) database (one local source is ITD's DSC which has records of all University IDs and the associated person's full name). If the ID is registered, the program simply procures the user's full name from there. Otherwise the front-end program needs to prompt the user for her/his full name. Finally, the program prompts the user for her/his preferred login name. At this point, the program has everything it needs to complete an Allocate Entry transaction (assume that the program is run with a default lifetime for all accounts created that term). If the desired name fails, the program lets the user know and offers to either try another user chosen name, use a computer-generated name, or abort the whole session. After several attempts, the program could refuse to try further names and just abort the session.

Assuming that the entry allocation completes OK, the program then goes ahead and initiates whatever further actions are needed to get the computer system(s) ready for the user.

Unaddressed Problems

- Single Point of Failure (will be extended to a replicated database later).
- Restricted UID space (some OS's are limited to 15-bit UIDs). Need to encourage vendors to move to at least 31-bits.
- Gigantic `/etc/passwd` files. Password files with thousands of entries can slow down commands such as "ls -l".

Conclusion

Uniqname offers system administrators an incremental way to convert their systems over to common login name and UID spaces and then to easily maintain them. It does so without affecting their autonomy over local issues such as login passwords, machine access authorization, account creation and deletion, and home directory placement. In fact, it does so without even assuming that such issues apply to the systems involved. Finally, use of Uniqname automatically generates both a common Kerberos database and an AFS 3.0 Protection database for all users registered with Uniqname. These in turn can be used as the foundation for a University-wide file system and other authenticated network services.

Further Information

Uniqname source and documentation may be obtained by anonymous FTP from:

ftp.ifs.umich.edu	~ftp/sysadm/uniqname
freebie.engin.umich.edu	/pub/uniqname

or via AFS from:

/afs/umich.edu/group/itd/ftp/sysadm/uniqname

Questions and comments about Uniqname can be addressed to uniqname_request@ifs.umich.edu.

References

[1] CCITT Blue Book, Volume VIII - Fascicle VIII.8, Data Communication Networks Directory - Recommendations X.500-X.521, "The Directory - Overview of Concepts, Models and Services", pp. 3-19 (Nov 88)

[2] J.G. Steiner, C. Neuman, J.I. Schiller, "Kerberos: An Authentication Service for Open Network Systems", Winter 1988 Usenix Conference Proceedings, pp. 191-211 (March 1988).

[3] J.H. Howard, "An Overview of the Andrew File Systems", Winter 1988 Usenix Conference Proceedings, pp. 23-26 (February 1988).

[4] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, "Design and Implementation of the Sun Network Filesystem", Summer 1985 Usenix Conference Proceedings, pp. 119-130 (June 1985)

Steve Mattson received his BSE in Computer Engineering from the University of Michigan in 1989. He continues to work for the College of Engineering coordinating several aspects of the user environment including accounts, printing, and most recently configuring the College's AFS cells. To get away from it all, he relaxes and works on planning his April wedding. Contact him at Computer Aided Engineering Network, College of Engineering; The University of Michigan; Ann Arbor, MI 48109-2092 or electronically at hobbes@caen.engin.umich.edu.

Yew-Hong Leong is a graduate student in Computer Science at the University of Michigan, Ann Arbor, where he received his undergraduate degree in Computer Engineering last Winter. He is currently on staff at the Center for Information Technology Integration working on the Institutional File System Project. Yew-Hong is involved in the engineering of the Uniqname package at the University. Contact him at Center for Information Technology Integration; The University of Michigan; Ann Arbor, MI 48109-4943 or electronically at leong@ifs.umich.edu.

Bill Doster is currently on staff at the Center for Information Technology Integration leading the Campus-Wide Filesystem Deployment Activities at the University of Michigan. He received his BSE in Computer Engineering from the University of Michigan, Ann Arbor. His interests include operating systems, file systems, security and scaling issues. Contact him at Center for Information Technology Integration; The University of Michigan; Ann Arbor, MI 48109-4943 or electronically at billdo@ifs.umich.edu.

LISA IV
October 17-19, 1990
Colo. Springs, Colorado

Kenneth Manheimer – NIST
Barry A. Warsaw – Century Computing
Stephen N. Clark – NIST
Walter Rowe – NIST

The *Depot*: A Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries

ABSTRACT

The *depot* is a coherent framework for distributing and administering non-OS-distribution UNIX applications across extensibly numerous and diverse computer platforms. It is designed to promote reliable sharing of the expertise and disk resources necessary to maintain elaborate software packages. It facilitates software installation, release, and maintenance across multiple platforms and diverse host configurations.

We have implemented the *depot* using conventional UNIX subsystems and resources combined with policies for coordinating them. This paper presents the specific aims, structure, and rationale of the *depot* framework in sufficient detail to facilitate its implementation elsewhere.

Keywords: *Depot*, UNIX, sharing, distributed file system, `/usr/local`, installation, third-party.

Introduction

Installing and administering third-party UNIX applications often requires significant investment of time and expertise, precious commodities in any organization. Duplicating this investment is usually not the most efficient way to distribute its benefits. Instead, it's much preferable to share the product of this investment in the form of stable, usable configurations, provided organizational and platform discrepancies between different machines can be overcome. The *depot* is a systematic organization for distributing the products of expert application maintainers' efforts in an efficient and unburdensome manner. The foundation of this system is a generalized framework for installation and maintenance of applications that accommodates distribution across multiple platforms in a versatile way.

With the greater distribution that this framework provides, reliability and change-release management become more critical. The *depot* has comprehensive provisions to reduce and sometimes eliminate difficulties inherent in greater operational interdependencies between hosts.

Depot Objectives

The *depot* provides a mechanism for distributing application installations across numerous machines. In order to be successful, it must accomplish this while meeting the following criteria:

- Generality: Accommodate diverse UNIX operating systems, hardware platforms,[1] and host configurations as well as diverse application packagings. Commercial, academic, and public domain packages each come with their own often elaborate installation methods and mechanisms and we need to accommodate them all.
- Robustness: Provide predictable and consistent services. Formalize procedures for staged release of new packages and new package versions.
- Scalability: Provide for incremental addition and commissioning of applications, clients,

[1]To date, the *depot* has been implemented only on various Sun Workstation architectures, but no essential mechanisms are Sun-specific. Our implementation makes extensive use of "conveniences" like Sun's *NIS* distributed administrative databases and Sun's *automount*[2]. *NIS* is becoming universally available, and automount capability is widely available as *amd*[3] for many UNIX and some non-UNIX platforms.

and servers to the extent that the underlying distributed filesystem allows.

- Reliability: Use reliable distribution mechanisms and support redundant fallback copies.
- Ease of use: Be easy to commission and employ. Avoid burdening either application administrators or users due to *depot* involvement.

What the *Depot* Is Not Intended to Do

The *depot* is not a project management system. Although it provides for staging software updates and releases, it is not intended for, nor is it particularly suited to, multi-agent source modification. It is best used for distributing software installation and upgrades, and not for software development itself.

The *depot* is not intended to replace usual conventions for software sharing but instead refines and complements their functions. The `/usr/local` directory hierarchy is typically used as a repository for installing non-core utilities and incidentals. Often this hierarchy is shared across clusters of hosts that are similar both in operating platform and in general organizational configuration and use. With the addition of the *depot*, `/usr/local` can continue to be used for those items specific to a distinct homogeneous cluster of machines. Those items warranting broader service across organizational and/or platform boundaries (and additional intrinsic rigors of modification and release) belong instead in the *depot*.

Depot Motivations

There is useful software that is not included in UNIX OS distributions.

Typical UNIX-based software development efforts require programming and other special-purpose tools that aren't part of the core OS distribution. For instance, at our site we use and maintain our own copies of freely available software such as the Gnu Project tools[7] and the X Window System[5], as well as numerous homebrew tools developed locally or floating around Usenet. We also use various third-party and "unbundled" software utilities, like commercial databases and publishing toolkits, that perform functions which are either not available in core OS distributions or only provided for in a rudimentary fashion.

Integrating such tools incurs substantial costs in expertise and other resources.

Expertise is expensive and must be applied efficiently. Non-OS software products, not delivered with OS distributions, often demand specialized expertise to maintain and accommodate them. Even well produced and packaged commercial products require disk space and expertise for their management. These products must be integrated with, and maintained in the context of, existing installations, which may already be specially tailored with diverse customizations.

Diversity can be an obstacle to sharing.

Large workstation-based computing sites generally consist of similarly configured subclusters of affiliated workstations. It is relatively straightforward to arrange to share distribution OS and other applications among the similarly configured members of one of these subclusters. (For example, it's quite common to find similar machines sharing network mounts or duplicates of a `/usr/local` filesystem that houses non-core applications.) However, differences between the configurations of machines in different clusters, or differences in OS revision, vendor, or hardware platform between machines that are otherwise similarly configured, thwart such direct approaches to sharing.

In particular, applications that can be prepared for diverse platforms usually require certain relationships between their executables, libraries, and other incidentals to be preserved across hosts. For instance, Gnu Emacs needs to know where to find its runtime lisp libraries, ancillary executables, and on-line documentation. More generally, many applications include runtime dynamically loaded libraries that need to be located in specific places for the applications to find them. Ad hoc sharing schemes developed for specific differences between specific machines will often fail to extend to other differences on other machines.

Generalized schemes may provide wider service at the expense of greater restrictions on client configurations. The challenge is to exploit sharing capabilities without imposing undue complexity or interference either on the existing individual host operating environments or on the experts administering the applications. In general, we don't want the savings in duplicated expertise required to manage the distributed software to be defeated by costs of accommodating or managing the distribution methods themselves. A good arrangement can avoid these pitfalls without compromising the benefits.

What the *Depot* Really Does

The *depot* is a network-filesystem based organization for sharing application installations across UNIX-based platforms. Most importantly, applications are easy to install and use from the depot. "Depotized" applications are arranged to be self-contained and structurally consistent across platforms so that internal relationships among application components are preserved regardless of the organization or platform of the client hosts. The *depot* avoids introducing undue dependencies between applications and their surrounding operating environments, or vice versa, and it does not interfere with intrinsic dependencies already present in an application.

Design Overview

Two abstract objectives have crucial influence over the shape of the *depot*:

- Provide transparent accommodation of multiple platforms
- Maximize generality; minimize dependence of depotized applications on the surrounding operating environment and on each other

Transparent accommodation of multiple platforms is accomplished by mapping from pan-platform server arrangements to platform-specific client arrangements.

While some third-party application packages accommodate multiple platforms with a single installation, most do not. Multi-platform installation and employment could be taken care of separately with platform-independent scripts. However, such scripts do not normally generalize from application to application. Indeed, it's difficult to arrange for the same script to take care of both installation and employment of even a single application. *Depot* structural arrangements instead transform an internal pan-platform arrangement of an application on servers to a public platform-specific arrangement on clients.

Separate directories are allocated in the pan-platform arrangement for the platform-specific portions of an application. Clients mount the entire pan-platform arrangement and then overmount the correct platform-specific components in a slot set aside for that purpose. Since the platform-specific components are effectively organized in the same way for all platforms and the platform-independent components are shared between all platforms, each client sees the same structural organization regardless of its platform. Only the platform-specific files themselves are different. This arrangement, as far as the client is concerned, looks like a configuration suited for installation and employment of the application on the client's own platform.

Arranging for simple mount schemes and minimizing dependence between *depot* applications and their operating environments dictates strong emphasis on self-containment.

Interdependencies between an application and its operating environment complicate the job of making the application widely available across diverse environments. In order to minimize this complexity we keep the arrangement of an application installation very self-contained. This self-containment is essential to avoid imposing unnecessary burdens on clients or application maintainers who use the *depot*. Dependencies of an application's installation on the structure of a client's operating environment are kept to a minimum, and, conversely, the *depot* design strenuously avoids imposing restrictions on the client's operating environment.[2]

Each depotized application is contained within a single directory hierarchy. The contents may be composed from mounts of scattered filesystems,[3] but they collectively look like a single hierarchy. The collection of *depot* applications is likewise contained within a single directory hierarchy. Those components that an application intrinsically requires to be established elsewhere in the operating environment are represented in the external locations by symbolic-link proxies that point to the actual components in their locations within the *depot* hierarchy.[4] (These links should be created by a script prepared as part of the process of incorporating an application into the *depot*, to ease commissioning of new clients.)

Implementation

The root of the *depot* hierarchy is located in the same place on clients and servers. All of the paths configured into depotized applications are prefaced by the path of the hierarchy's root, so a short one is preferable. We have our *depot* root located at `/depot`.

A *depot* installation of an application has two principal aspects: the arrangement of disk storage for the pan-platform components of the application, and the public interface to it. The platform-specific public interface is implemented on every client that subscribes to the *depot*, and we will refer to it as the "client view". The client view is composed, using *NFS* mounts, loopback mounts, or symbolic links, from the pan-platform arrangement, which we will refer to as the "origin view".

A host that serves as an origin for an application (or for a piece of it) usually also makes use of the application as a client and so employs both origin and client views. (The converse, however, is not true: the majority of clients do not serve as application origins.) Together with the mechanisms that we use to compose the client view from the origin view, this requires the *depot* location of the client view to be different from the *depot* location of the origin view.

[2]Note that *depot* packagings for an application *may* include shortcuts that involve nonessential client dependencies, just so long as their functionality is available in other, non-constraining ways.

[3]For instance, sometimes filesystem service of an application's platform-specific components is distributed among different hosts, with each host serving only the components which are specific to its own platform.

[4]X11 under SunOS 4 contains an example of an application with components that need to be located outside of the *depot* hierarchy. *Xterm* depends on a dynamic library which must be located in either `/usr/lib` or `/usr/local/lib` for setuid-authorization purposes.

We will first detail the arrangement of the origin view. Next, we will describe the public interface provided by the client view, and finally, the mapping from the origin view which is used to implement the client view.

The Origin View

The origin view contains all of an application's components, including a single copy of each of the platform-independent components and a copy of each of the platform-dependent components for each of the supported platforms. It is not intended, however, to be directly usable for either installation or execution of the application - this is the role of the client view.

Within the *depot* root, application origins are located in subdirectories whose pathnames begin with either `/depot/.primary` or `/depot/.develop`.[5] These two directories differ only in the way they are used; their internal organizations are identical. Fully released, in-service application copies are situated in `.primary` directories. The `.develop` directories provide private, temporary work areas in which to perform *depot* application builds or experiment with changes without affecting other users. (See "Isolating Release Preparations for Upgrades" below for more details.)

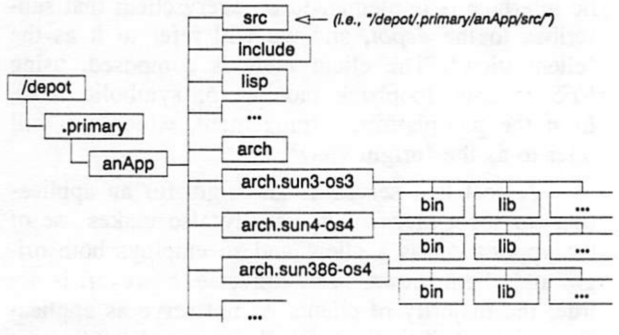

Figure 1 – Origin View of *anApp*

Figure 1 shows a portion of the origin arrangement of a fictional application named *anApp*. Each box represents a directory (or collection of directories, in the case of the boxes containing ellipses). Sibling directories on the path above `/depot/.primary/anApp` are ignored.

- The `src` directory contains the source distribution for *anApp*.
- The `include` and `lisp` directories are typical examples of subdirectories containing platform-independent components.
- The `arch` directory is a stub necessary for use in constructing the client view.
- `arch.sun3-os3`, `arch.sun4-os4`, and `arch.sun386-os4` are typical examples of directories which contain platform-specific components. They commonly have subdirectories `bin` (for public executables) and `lib` (for public and internal object libraries).

[5]The dot '.' prefixes are not so much for the ostensible (and rather thin) UNIX purpose of "hiding" these directories, but rather for the sake of distinguishing them from the other contents of the directory by clustering them together at the front of *ls* listings.

It is common to have separate `lib` directories for platform-independent and for platform-dependent components. The platform-independent `lib` might contain ASCII text files like default and "rc" configuration files, skeleton files for code generators, etc., while the platform-dependent `lib` would hold object code libraries and byte-order-sensitive files like fonts.

The name of each platform-specific directory distinguishes the platform to which it belongs. It is only necessary to distinguish between fundamental OS, hardware executable format, or byte-order incompatibilities.

Each platform-specific directory name starts with the prefix "arch."[6] The next few letters indicate the hardware architecture of the supported platform. Finally, the string "-os" is concatenated with a string that indicates the supported operating system. Thus, for example, `arch.sun4-os4` denotes the directory for software specific to Sun SPARC ("Sun4") architectures running SunOS 4.

Depot servers need to grant at least remote read-access privileges for clients to mount the origin directories. Since compilation and installation are done within the client view, clients used for *depot* administration must have read/write privileges. We use "read-mostly"[7] together with root-access designations to grant suitable privileges to the specific clients that will be used for building while restricting all other clients to read-only. This assures that only authorized clients can be used to make changes to the applications.

The Abstract Client View

A client view of an application is a platform-specific arrangement employed for both administration and public use of the application on a particular machine. Composed from the pan-platform origin view using *NFS* mounts, loopback mounts, or symbolic links, in the abstract the client view looks like a dedicated installation of the application for its host platform.

All path references for the application, whether for internal configuration or for public access, use paths dictated by this abstract arrangement. Thus it provides both the public interface to the application and the internal interface between its components.

[6]"arch." is a holdover from early *depot* days; "plat." or "platform." probably would have been more appropriate.
[7]SunOS *exportfs* (8) man page[1].

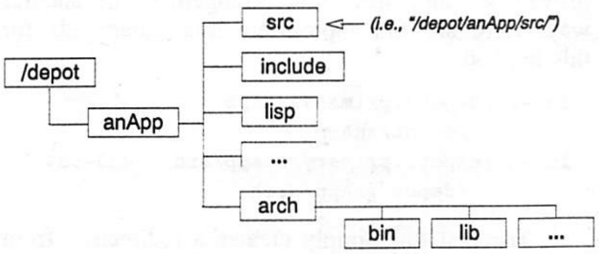

Figure 2 – Abstract Client View of *anApp*

The Origin-Client Mapping

The mapping between the origin view and the client view is the crux of the *depot* scheme. The client view is composed by mounting the server arrangement and then overmounting the suitable architecture onto the empty `arch` directory. In the absence of *automount* and loopback mounts origin servers can use symbolic links to achieve this client mapping locally. (The implementation of the mapping is explained in detail in "Implementing the Client View", below.)

Using our example, the *anApp* root origin would be `/depot/.primary/anApp` on some host. This origin is mapped to `/depot/anApp` on the client. Next, the particular `/depot/.primary/anApp/arch.<arch>-<os>` directory suited to the client platform is mapped to the `/depot/anApp/arch` stub directory, provided specifically for this purpose. As a result of this mapping the `arch` directory on the client effectively contains the platform-specific components of the application required by the client's platform.

The resulting arrangement on the client is illustrated in Figure 3. It shows the typical arrangement of a *depot* client, including the location of the origin root directory on those clients that also serve as *anApp* origins. Each directory is represented by a box whose shading indicates the role it plays in the arrangement and in the mapping.

- **Plain Local** (`/depot`) are regular directories in the root of the local file system.
- **Local, only on Origin servers** (`.primary`, `.develop`) are directories that are present on clients only if they happen to be origin servers. ("The Origin View" section, above, details their contents.)
- **First Redirects** (`anApp` and its subdirectories) are established by a mount of or link to the root directory of the *anApp* application on the origin server (`/depot/.primary/anApp`).
- **Overlaid Redirects** (`arch` and its subdirectories) are established with a second mount from the `anApp` origin hierarchy onto the empty `arch` directory. The mount maps the particular platform-specific directory for the host (in this example, `arch.sun4-os4`) into the `arch` directory of the client view.
- **Other Redirect Stubs** (`X11, yAp, zAp, ...`) are shown simply to illustrate that clients may subscribe to numerous applications. Structural details are not shown but would follow the same principles illustrated by `anApp`.

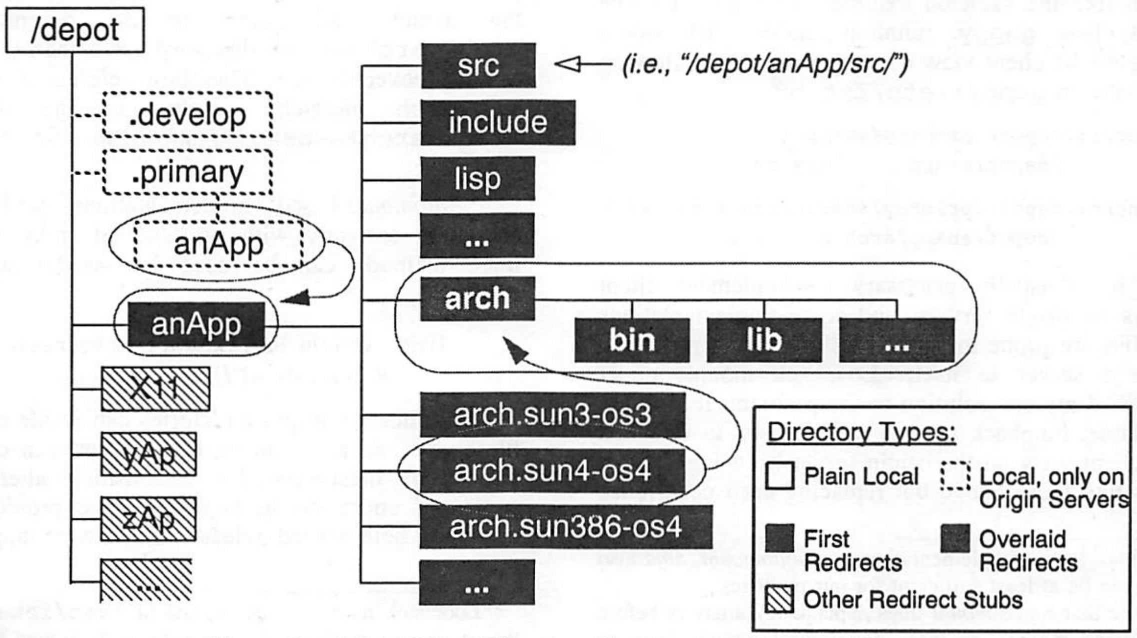

Figure 3 – Origin/Client Mapping of *anApp* on a Sun4-os4 Host

As mentioned above, the only important `arch*` directory in the client view is `arch`. This directory effectively contains the platform-specific components of the application, and the `arch.<arch>-<os>` directories are ignored.

Both application configuration and public references to application components should resolve to the `arch` directory. Public access to any platform-specific components should either refer directly to `/depot/anApp/arch` subdirectories or get to them via symbolic link proxies. In this way diverse clients use what appears to be the same structure to resolve both platform-dependent and independent application components, and both installation and employment of an application use the same paths regardless of the client's platform.

Implementing the Client View

Sun's *automount* significantly simplifies implementation of the origin/client mapping.[8] It provides the means to systematize and distribute mount configurations via a networked administrative database (*NIS*). It also accounts for special requirements of hosts that serve as both origins and clients of an application. In Appendix I we include a representative *automount* map to help illustrate how to use *automount* for *depot* purposes. Below we detail the non-*automount* procedure for implementing our *anApp* example, both for the sake of clarifying the origin/client mapping and to show how to implement it when *automount* won't be used.

We'll use the syntax of Sun utilities for our example. The Sun4 host `honcho` running SunOS 4.1 will serve the application *anApp* from the origin directory `honcho:/depot/.primary/anApp`. (*anApp*'s directory structure on `honcho` would look much like the skeleton exhibited in Figure 1.) The Sun3 client `guppy`, running SunOS 4.0.3, would compose its client view of *anApp* with the following two lines in `guppy:/etc/fstab`:[9]

```
honcho:/depot/.primary/anApp \
        /depot/anApp       nfs rw 0 0

honcho:/depot/.primary/anApp/arch.sun3-os4 \
        /depot/anApp/arch nfs rw 0 0
```

It is usually necessary to implement client views on origin servers, and some implementations of NFS are prone to serious failures when mounting from a server to itself. Loopback mounts under SunOS 4 are one solution to the problem. If they are available, loopback mounts can be used to establish the client view on the origin server by using `fstab` lines like those above but replacing each occurrence of "nfs" with "lo".[10] Symbolic links can be used instead to implement this arrangement in another way. Here are the appropriate link commands for this method:

```
ln -s /depot/.primary/anApp \
        /depot/anApp
ln -s /depot/.primary/anApp/arch.sun3-os4 \
        /depot/anApp/arch
```

The first link simply creates a redirection from the client view `/depot/anApp` directory to the application origin at `/depot/.primary/anApp`. The second link creates a redirection from the client view `/depot/anApp/arch` directory (which is identically the origin view directory `/depot/.primary/anApp/arch`, as a result of the previous link) to the platform-specific origin directory `/depot/.primary/anApp/arch.sun4-os4`. Note that if the platform-specific stub, `/depot/.primary/anApp/arch`, already exists, the second link will not be properly created, but will instead be placed inside this stub directory.

There is a somewhat obscure complication in the link scheme which turns out not to be a problem but which bears explaining nonetheless. Since the second link is actually established in the origin view as the `arch` redirection, it's seen by all clients that mount this filesystem. The question is how this affects a client that uses mount to redirect the arch directory (in this case the link) to the appropriate platform-specific directory.

Since the link is resolved at mount time, the desired `arch.<arch>-<os>` directory is mounted over whichever platform-specific directory this link points to. The `arch` link is therefore not covered by the mount, and points to the overmounted `arch.<arch>-<os>` directory rather than the shadowed (covered) one. Therefore references within the `arch` hierarchy resolve to the desired `arch.<arch>-<os>` platform-specific components.

Automount-based implementations will also cooperate correctly with origin/client links, so all three methods can be used in parallel without conflict.

Using Origin Redundancy to Increase Reliability of *Depot* Services

Application origin directories can reside on any fileservers and they can be divided between combinations of fileservers. By establishing alternative copies of entire origin hierarchies we provide the basis for both staged release of software upgrades

[8]Although our implementation uses *automount*, *amd* also seems to be at least sufficient for our purposes.

[9]Note that Sun's *mount* does dependency analysis before processing mounts, so these hierarchical mounts pose no problem.

[10]Loopback mounts must be last in `/etc/fstab`; see the warnings section of the SunOS 4 *mount*(8) man page[1].

and fallback redundancy to increase reliability by reducing critical points of failure.

Multiple `.primary` Origins Provide Redundancy

By creating multiple `.primary` origin hierarchies for crucial applications we are able to achieve distributed loading of the origin servers and, perhaps more importantly, provide fallback service in the event of a fileserver failure.

Since there are multiple servers for complete origin copies, a client will have alternates from which it can get the software if an origin server goes down. At worst a client will need to reboot to free itself from a locked-in mount. (A mount can become locked in when an active executable can't be completely terminated because it's hanging on a disk read from the defunct filesystem.) Once the client is freed it can redirect its mounts to surviving alternate servers. With *automount* the rebinding process is automatic, though locked-in mounts may prevent rebinding, so that reboot may sometimes still be required.

Isolating Release Preparations for Upgrades

For tools that are perpetually in use, like Emacs and X, it's important to minimize down time. The building and testing phases inherent in controlled releases of new versions can be time consuming. By arranging for a client to use its own copy of an application origin the release can be thoroughly prepared in isolation.

Using one of multiple `.primary` origins is not suitable for this purpose. Publicly enlisted alternate `.primary`'s must be kept synchronized and it is awkward (and usually undesirable) to quickly remove a publicly enlisted origin from service. Instead we establish distinguished `/depot/.develop`[11] copies specifically for preparing the release, doing the build and testing in isolation from the rest of the world. Once the release is prepared it is announced and migrated as a whole to the `.primary`'s. (The `.develop` versions can then be deleted, although they may be useful as placeholders to reserve disk space for the next release cycle.)

Some Incidentals on Implementing Redundancy

We use a special *automount* map akin to the Sun "`-hosts`" map that allows us to see the entire origin structure of all servers to facilitate copying the contents of the `.develop` origins to the corresponding `.primary` origins. This is especially useful when we use multiple `.primary` origins for an application to provide redundancy.

[11]Another holdover from initial development which unfortunately implies something other than what we mean. This might more appropriately be called something like `.aside` or `.scratch`.

We have resolved some formal policies about management of the `.primary` and `.develop` application origins in order to facilitate cross-divisional use of the *depot*. Most importantly, multiple `.primary` origins for an application are guaranteed to be held as consistently identical as can be managed. This is necessary to ensure that clients can rely on identical service from any of the `.primary` copies of the application. It is crucial when using *automount* with multiple primaries because *automount* does not necessarily use the same host for **first redirects** and **overlay redirects**, and will combine platform-independent and platform-specific components of an application from separate servers.

Also, it's important to identify managers for each application who will at least coordinate additions and upgrades to it. We stipulate that any changes of applications must be arranged with the designated application administrator, and any potentially disruptive releases to `.primary` origins should be done with the direct involvement of the administrator. Furthermore, as with any system changes that impact users, any potentially disruptive changes should be scheduled to the satisfaction of the range of clients using the applications.

Results

We have been developing the *depot* for about a year now and have been using it in near final form for the last half year. Its use spans two major organizational divisions of our laboratory, and will soon include a third. We use it to serve numerous applications to a contingent of more than one hundred workstations, at one point including seven distinct operating "clusters", nine comprehensive file servers, and three major OS versions.

Most dramatically, both divisions have a larger repertoire of better maintained utilities thanks to their availability through the *depot*. We use major research and academic programming tools including X, NeWS, Gnu, InterViews, and Usenet news facilities, and numerous commercial products including FrameMaker, Saber-C, Parasolid, Hoops, and Allegro Common Lisp. We have consistently maintained an up-to-date repertoire of all of these tools across two major OS releases (Sun OS 3.5 and 4.0/4.1) and three different hardware architectures (Sun-3/68020, Sun-386i, and Sun-4/SPARC) with only a single central administrator for each application (two for Gnu - one for Emacs and one for the rest) providing service for both divisions.

Redundant origin hierarchies are big wins. By dedicating some disk space to additional origin hierarchies, reliability can clearly be enhanced well beyond what would be available with a single origin. In the case where multiple clusters are sharing services, redundant hierarchies may not even require extra disk space - it is likely that each cluster already maintains its own copy of an application, so that all that is required is to implement *depot* disciplines on the various hosts.

The consistency of depotized application distribution makes one copy interchangeable with another, while separately managed versions usually are not trivially interchangeable for the reasons cited above (see "Diversity can be an obstacle to sharing"). Large groups of clients can be served by relatively few copies, so the returns improve up to some fairly high server or/and network loading (or even connectivity) saturation point as scale increases.

By establishing duplicate primaries for important *depot* applications on mutually independent cluster servers we've achieved much better uptime. In particular, because of the immediate interchangeability of the duplicated applications, we can have one server off-line (either intentionally or due to a system failure) and only those machines dependent on boot services or on applications not incorporated into the *depot* are incapacitated. During four separate major system failures over the past year we reduced what would have been down-time for some major applications (X, Emacs, FrameMaker) for at least thirty machines (and up to sixty machines, depending on which division's machine was hit) to down-time for only a maximum of ten dependent boot-clients. Considering that the repairs on one of those occasions stretched out to over a week, that constitutes a major reduction in lost work-hours.

Perhaps the most outstanding sign of the success of the *depot* is the degree to which the respective divisions' managements allow this cross-dedication of talent to each other's facilities. We feel that the only reason we are able to "get away" with this is because they recognize, as do we, that we're all getting more comprehensive and thorough service with less invested effort and greater ease of use than we did prior to the commissioning of the *depot*. And that was while we were still developing it ...

Summary

The *depot* provides a framework for installing arbitrary software, including third-party and custom applications, to accommodate diverse platforms. The structural arrangement of an application's installation is consistent from one platform to another, allowing the same usage and installation path across platforms. Applications pre-packaged with multi-platform accommodations are simply installed without fuss. Commissioning an application's *depot* installation requires no more finagling than does adapting multiple copies of the same configuration for installation on multiple standalone machines, and usually requires less effort if the standalone machines don't happen to be identically arranged.

All components of the *depot* except for the inherently public application components (the user interface) are confined to a single directory hierarchy on any client's file system. Simple relationships among the actual installed components hold regardless of the host file system environment. Even the external interface is implemented as a simple, reproducible, and platform-independent entity. Thus commissioning an application in the *depot* usually entails establishing a small set of filesystem mounts, establishing the application-mandated hooks if any, optionally establishing symbolic link surrogates for the external interface for access, and creating a script to automatically create all of the necessary external links identified in this process (this is to ease the commissioning of new clients). This almost always winds up being even simpler and cleaner than it sounds.

The consistent organization of *depot* sharing allows redundancy to be used directly to increase reliability. As scale increases the returns increase, up to the saturation point of the media (fileserver hosts, *NFS*, and/or network).

It is important to note potential problems that the *depot* avoids. It depends only on conventional UNIX utilities and imposes minimal overhead on the application servers, maintainers, users, and client systems, providing a robust basis for multi-platform support of diverse utilities. It does not interpose clumsy interfaces for installing or accessing platform-specific components of an application, relying instead on remounts which are almost entirely transparent to both application management and the user clientele.

Unresolved Issues and Other Work

- Applications with installed components that are not strictly partitioned from their source distribution require extra finagling for installation in the *depot*. For instance, X11r3's *imake*[6] mechanism required some extra effort in order to establish this partitioning, though X11r4 has solved that problem with the introduction of *xmkmf*. Gnu Emacs also exhibits the problem. It uses the distribution `etc` directory for ancillary executable components that are necessary both for build and operation of the application. It is necessary to build some custom scripts in order to implement the partitioning for Gnu Emacs. We think it may be reasonable to consider this partition between source distributions and built releases as one criteria of a "good" installation mechanism, but have to evaluate this further.
- We need to implement the *depot* on other non-Sun machines. While we have small numbers of various other UNIX platforms around, including Silicon Graphics, DEC, and IBM, none of the active *depot* development personnel are responsible for those machines. Now that we have reached a fairly stable

framework we intend to branch out a bit.

- We need to investigate newly available technologies, e.g. "translucent" file systems[4], and evaluate how we can use them to improve on the simplicity and transparency of the system.

Acknowledgments & Disclaimer

This work was jointly funded by the NIST Automated Manufacturing Research Facility (AMRF, project 734-3385) and Scientific and Technical Research Services (STRS, project 734-3106).

The *depot* scheme was initially conceived and designed by Ken Manheimer. Barry Warsaw and Ken refined the initial design. Barry implemented a prototype layout and Ken implemented the initial use of the overmounting scheme. Barry and Steve Clark developed specific methods for managing the Gnu software package as a whole. Walter Rowe did some similar work for sundry X tools. The initial layout, along with the conception of the *depot* in general, was further refined and resolved by the concerted efforts of all of the authors.

We are indebted to our collective management and to our numerous users in the Factory Automation Systems Division and the Robot Systems Division at NIST, who on numerous occasions had to put up with the growing and shaking-out pains of the progressively developing system. In particular, thanks to Scott Paisley, another local system manager, for valuable input and assistance, and to local guru Don Libes, who provided important criticism and insight while we were developing the depot and who encouraged us to submit and write this paper. (He was also the only person who had the guts to read early drafts of this paper.)

No approval or endorsement of any commercial product by the National Institute of Standards and Technology is intended or implied.

Appendix - A Representative Automount Map

The automount fragment depicted in Figure 4 illustrates some nuances of Sun's *automount*, particularly the combination of hierarchical mounts and alternative servers for a common hierarchy.

Note that hierarchical automounts composed from alternative servers can be and often are realized with components from both servers. For example, it is not unusual to find the `/depot/gnu` directory mounted according to the above fragment to come from the host `imp` and the `/depot/gnu/arch` directory to be mounted from `dip`. For this and other reasons it is imperative that the alternative origins be held in strict synchronization.

References

[1] Sun Microsystems Incorporated, *SunOS 4.1 Reference Manual.*

[2] Sun Microsystems *SunOS 4.1 System Administration Guide* or *SunOS 4.0.3 System Administration Addenda* is an essential supplement to the *man* pages.

[3] Jan-Simon Pendry, "Amd - An Automounter", Department of Computing, Imperial College, London, England, 1989.

[4] Sun Microsystems Incorporated, "TFS", *SunOS 4.1 Reference Manual*, Vol 2, p. 1494.

[5] Scheiffler, R.W. and J. Gettys, "The X Window System", *ACM Transactions on Graphics* Vol. 5, No. 2, April 1986, pp. 79-109.

[6] Jim Fulton, "Configuration Management in the X Window System", The MIT X Consortium, MIT, Cambridge, MA, 1989.

```
## Note: We *cannot* include entries that cause a
#  dir to go on top of itself.
#target root    dir/opts    <Sys>:<path> origin
#-----------    --------    -------------------
/depot/autotabs / -ro       elf:/depot/.primary/autotabs
/depot/sundry   /           elf:/depot/.primary/sundry \
                /arch       elf:/depot/.primary/sundry/arch.sun3-os4
/depot/X        /           imp:/depot/.primary/X \
                            dip:/depot/.primary/X \
                /arch       imp:/depot/.primary/X/arch.sun3-os4 \
                            dip:/depot/.primary/X/arch.sun3-os4 \
                /src        dip:/depot/.develop/X/src
/depot/gnu      /           dip:/depot/.primary/gnu \
                            imp:/depot/.primary/gnu \
                /arch       dip:/depot/.primary/gnu/arch.sun3-os4 \
                            imp:/depot/.primary/gnu/arch.sun3-os4
```

Figure 4 – Automount Fragment

[7] Available from The Free Software Foundation of Cambridge, Massachusetts, further information is available via electronic mail on the Internet from gnu@prep.ai.mit.edu.

Ken Manheimer works as UNIX Systems Support Manager in the Factory Automation Systems division at National Institute of Standards and Technology, where he has shepherded the growth of his divisions UNIX computing from four Sun 1's (and Eunice on a VAX) to seventy+ UNIX systems. He received a B.A. in Computer Science from Hampshire College. Reach him at NIST; Bldg 220, Rm A127; Gaithersburg, MD 20899 or electronically at klm@cme.nist.gov.

Barry A. Warsaw has just recently joined Century Computing, Inc. as a Data Systems Engineer, where he will be working on an online retrieval system for the National Library of Medicine. Formerly with NIST, he was at times system manager for the Robot Systems Division network of UNIX machines, and developer of user interfaces for robotic and automated machine control systems. He received a B.S. in Computer Science from the University of Maryland. Reach him at Century Computing; 1014 West Street; Laurel, MD 20707 or electronically at baw@fox.gsfc.nasa.gov.

Stephen N. Clark has never been a system administrator in his life, falling instead into the amorphous category of "knowledgeable user." He is currently working on tools for building schema-driven applications in support of the National PDES Testbed at NIST. He received an Sc.B. in Math and Computer Science from Brown University. Reach him at NIST; Bldg 220, Rm A127; Gaithersburg, MD 20899 or electronically at clark@cme.nist.gov.

Walter Rowe is currently the System Administrator for the Robot Systems Division of the NIST, where he maintains a network of 30 Sun workstations. He received a BS in Computer Science from Tennessee Technological University and is currently working on a MS in Computer Science at the Johns Hopkins University in Gaithersburg, Maryland. Reach him at NIST; Bldg 220, Rm B124; Gaithersburg, MD 20899 or electronically at rowe@cme.nist.gov.

LISA IV
October 17-19, 1990
Colo. Springs, Colorado

Kevin C. Smallwood – Purdue University Computing Center

Guidelines and Tools for Software Maintenance in a Production Environment

ABSTRACT

The Computing Center is a service organization at Purdue University; the UNIX Systems Programming Group's charter clearly states that "the customer comes first." The Purdue University Computing Center runs stable, production systems for students and researchers alike. A production environment often implies little or no change to system software. While it is true that modifications in operating system software happen much slower in production environments, changes do and will happen to keep reasonably current with bug and security fixes and newer technologies. We have developed a set of guidelines and tools to allow modifications to system software while minimizing the impact on our customers.

The Purdue University Computing Center

The Purdue University Computing Center provides computing resources to a variety of customers. We have two VAX 11/780 systems running 4.3BSD UNIX, one VAX 8800 system running Ultrix, four Sequent Symmetry systems running DYNIX, three Sun-3/280 systems running SunOS 4.0.1, and one ETA-10P* running ETA System V UNIX. We provide computing resources for instructional and research computing for many departments on campus; however, we are not the sole provider of computing services on campus.

Instructors, students and researchers are not too concerned with new system features or the latest *whiz-bang* compiler; they typically want to use the computing resources as a "tool" to get their work done, such as teaching, programming assignments, or research data analysis.

The West Lafayette campus of Purdue has a yearly enrollment of 35,000 undergraduate students and 10,000 graduate students. It is normal for a student to use the Computing Center's computing resources at least once during his or her academic career. Students have a low tolerance for system crashes or ill-working utilities when their programming assignments are due; system stability is very important. The same low tolerance applies to instructors and researchers.

The Computing Center's UNIX Systems Programming Group is composed of nine full-time UNIX systems programmers and up to five student part-time programmers. The Computing Center's Super Computing Systems Programming Group and the UNIX Systems Group are jointly responsible for the ETA System V UNIX software. That means that another six full-time systems programmers may be working on parts of the software. In an effort to promote creativity and innovation, we do not saddle the programming staff with volumes of policies, procedures and rules. Instead, the UNIX Systems Group has developed a few common sense guidelines for installing new software, installing changes to software, and phasing out old software in a way that will minimize the impact on our customers. Furthermore, we have developed a few tools to aid and enforce these guidelines for the protection of everyone.

Problems We Have Encountered And Solutions We Have Developed

Software Testing Environment

One problem associated with developing new software or applying bug and security fixes or new features to existing software is that of testing the new software in a *production* environment without subjecting our customers to instability. The method we have used to get around this is to require that new software (including new changes to existing software) be installed in "/usr/new/bin," "/usr/new/etc," or "/usr/new/lib" for a period of at least two weeks before installing them in production directories. After the programmer has done as much testing as can be done on his own, the binaries are installed in the appropriate subdirectories of "/usr/new." The new binaries are owned by the programmer unless they must run *setuid* as something else (e.g., *root).*

The source code for new products is installed in the appropriate subdirectories of "/usr/src/new" so that other programmers in the Group can examine the code if an emergency should arise, and the principal programmer is not available. We use RCS[1] for maintaining changes to existing software.

After the new binaries and sources are installed, the programmer posts a network news article to a local news group specifically for Computing Center Staff. The news article describes the new software or changes and the location of the new programs. "/usr/new/bin" does not exist in the default **PATH** for production shells at the Computing Center; by default, our typical customer will not execute the new programs. All the UNIX Systems Group and many Computing Center Staff have volunteered to have "/usr/new/bin" (and "/usr/new/etc" for systems staff) first in their **PATH**'s. The Computing Center Staff can be the most critical users of new products and changes to working software. The goal is to have the last few bugs exposed and fixed while running the new software in a controlled environment, where informed and tolerant users can be found.

Once a week, we run (via *cron*(8)) a program called *tickle*. The *tickle* program's main function is to look in "/usr/new" and "/usr/src/new" for files that are more than two weeks old (the default) and send reminder mail to the owner of the files[2] that it is time to consider installing the files in the production directories. This is only advisory; conditions may exist where it is not wise to install the new software until a later date. *Tickle* prevents software from being installed somewhere on a *temporary* basis and forgotten forever.

However, some software, such as system daemons, cannot easily be tested this way. These are *guidelines*. The final decision is given to the programmer. Sometimes, only a simple change is made to an error message; it is counterproductive to subject competent staff to bureaucratic procedures. Furthermore, perhaps a package may have been running successfully on a system in another department at Purdue University, or a change may be required immediately to fix a serious flaw or security hole in an existing program; again, installing these directly into the production environment is better. Exceptions will exist and the last things needed are rigid policies keeping people from doing their jobs.

[1]Revision Control System by Walter F. Tichy, now available from the Free Software Foundation, Inc.

[2]This is the main reason the principal programmer should own the files in "/usr/new" and "/usr/src/new;" reminder mail to *root* gets annoying quickly.

Safe Software Installation

When we determine that new software is ready to be moved into the production environment, we follow guidelines that minimize the impact on the customer. As with so many other sites, we started out using the *install*(1) script that was provided with 4.2BSD UNIX. This *install* program is a shell script.

After installing a few utilities and then discovering that latent bugs had survived the "/usr/new" test bed,[3] we wanted a way to quickly recover the old binary.[4] While we could back-out the source code changes, recompile a binary using the old source code, and then install the previous working binary; sometimes these *minutes* will seem like **hours** when you have a large public terminal room of angry students making references to your ancestors. It is times like these when you really want the previously existing binary available so you can quickly install it back into the production environment.

We decided that when we installed a new binary, we would *back-up* the existing binary. To do this, we create subdirectories called "OLD" and move the existing binary there before installing the new binary. Some sites move an existing binary to its name with a ".old" or ".orig" extension (e.g., "/bin/ls.old"). We felt that this cluttered the name space,[5] but more importantly, it might get our customers dependent on ".old" or ".orig" binaries. If we really want to keep old versions of production programs, we put them in "/usr/old/bin." By retaining the existing binary in an "OLD" subdirectory, we can easily and quickly back-out an installation if errors are found. A new version of *install*(1L) was created. This started as extensions to the shell script provided by Berkeley, but quickly turned into a C program for performance and extension purposes.

We then realized that if we did not have a method to periodically clean out the "OLD" subdirectories, we would eventually be short of disk space. The *purge* program was born. *Purge* used *find*(1) to traverse certain system directories[6] and find all the "OLD" subdirectories. It then used this list of "OLD" subdirectories to remove the files that were older than two weeks (by default). We decided that if problems had not shown up in two weeks of production, then nothing major was likely wrong. Because the shell script version of *purge* used a local option to *find*(1), did not handle symbolic or

[3]Yes, it will happen!

[4]"Binary" could be a script or compiled object code program.

[5]Although it is doubtful that anyone would really want to name a production program "ls.old" or "ls.orig."

[6]Realize, we had to be careful to not traverse our customers' directories since the directory name "OLD" was not really a *sacred* name; anyone could (and did) use it.

hard links well, and was not generally useful to our customers, it was recently replaced by a C program version.

If we *install*'ed the same program more than once in a short period of time (i.e., less than two weeks), we would lose the previous binary. We extended *install* to save previous "OLD" files by attaching the process identifier to the file name. That is, if you *install*'ed a new version of "/bin/ls" yesterday, then "/bin/OLD/ls" exists. *Install* moves "/bin/OLD/ls" to "/bin/OLD/ls<PID>," where "<PID>" represents the process identifier of the *install* process. "/bin/ls" is then backed up by creating a hard-link from "/bin/ls" named "/bin/OLD/ls." To insure that the new binary is in the same file system as its destination directory[7] so that the *rename*(2)[8] system call can be used for the final installation, *install* copies or moves (depending on the option given to *install)* the new binary to the "OLD" subdirectory under the destination directory with a unique file name (i.e., "/bin/OLD/<*unique-name*>"). Finally, the existing target (i.e., "/bin/ls") is *unlink*(2)'ed, leaving the back-up copy in "/bin/OLD/ls," and the new binary is *rename*()'ed to the target (i.e., "/bin/OLD/<*unique-name*>" is renamed "/bin/ls"). The saving of previous old versions of binaries allows us to restore a known working version even if a few other *newer* versions of the binary had been *install*'ed.

While the above method of using the "OLD/<*unique-name*>" file seems unnecessary, using the *rename()* system call (where available) opens only a very small window where a system crash would leave the directory contents in an inconsistent state. Further, the use of this third (<*unique-name*>) file name lets you use *install* to re-install the old binary; effectively swapping the two files (i.e.,

install /bin/OLD/ls /bin/ls

implies: "/bin/OLD/ls" → "/bin/ls" while "/bin/ls" → "/bin/OLD/ls" as part of the normal installation back-up process). We have used this many times!

Secure Software Installation

We quickly realized that the ownerships, group ownerships and modes of system files were very important. If a program needs to run *setuid root,* then installing that program without the *setuid root* permission impacts the users of that program. So, even a totally bug-free program must have the proper ownerships, group ownerships and modes to be functional.

Again, our local *install* program was enhanced. We established default owner, group and mode for files being *install*'ed without any overriding options. If the target file existed, we used the owner, group and modes of the target file unless overriding options were given in the invocation of the *install* command. We also dropped any *setuid* or *setgid* permissions on the backed up ("OLD") version of the file.

We saw the need to insure that files are *install*'ed with the proper owners, group and modes in the first place; if a file was *install*'ed incorrectly the first time, inheriting bad attributes would only propagate the problem. *Install* now reads a configuration file, *install.cf,* to see what actions and safeguards it should enforce. *Install* scans the configuration file for a *sh*(1) *glob* expression that matches the file you plan to *install.* If it finds a match, it then enforces certain rules about the ownership, group ownership, modes, and whether the file should be run through *strip*(1) or *ranlib*(1).

While many "Makefile" files are used to install files, programmers will often install parts of a software package interactively; furthermore, there is no reason to believe that the "Makefile" is correct. The *install* configuration file guards against simple typographical errors in important system file installations. For example, we once found a case where a programmer typed (as *root):*

install -m 7555 -o root -g wheel sh /bin

Note the extra "5" in the mode. Yes, /bin/sh was made *setuid root* and *setgid wheel!* This can happen (and did!). Our installation configuration file no longer allows this to happen.

Our version of *install* also reports on the existence of hard-links to an existing file. For example, if you are *install*'ing a new version of "/usr/ucb/vi," you may have forgotten about the five other links to it. If someone invokes "/usr/ucb/view," he or she may not get your new version of the editor. *Install* notices these links (when it is backing-up the existing file) and warns you about the existence of the links. As an extension, we allow you to specify a list of hard- or symbolic-links with options to *install.*

Safety And Security After Installation

Many papers have been written about programs that check the file ownerships, group ownerships and modes for security reasons. Some programs check dates and sizes of programs for unauthorized changes. We run some of these programs to enhance the security at our site. However, we wish to ensure that our file ownerships, group ownerships and modes are in a *pristine* state. We already have an installation configuration file in place, but this only helps when we use *install* to install new files.

[7]This also implies that "OLD" subdirectories cannot be file system mount points or symbolic links to directories on other file systems.

[8]If available. Unfortunately, the *rename*() system call is not available on all systems.

A new program called *instck*(1L) (install check) was written. *Instck* does what *install* does when it installs new files, but, no files are installed. *Instck* generates reports and an effort is made to correct violations. A useful feature of *instck* is that it can generate a reasonable checklist configuration file that you could use as your first-draft[9] *install.cf* file.

Keeping Everyone Informed

We learned early that our customers do not like surprises (instabilities). We established a guideline of using the news system to keep many of our customers informed of changes. If a change is considered *major*, we refer to it in the "/etc/motd" file. We limit the noise level in "/etc/motd"[10]; therefore, if the change requires several lines to accurately describe it (and many do), we refer people to a file kept in the "/usr/news" directory (not to be confused with the network news system). For example, if we plan to install a new version of the *MH* system, we use these lines in "/etc/motd:"

MH Users: We will be upgrading to MH 6.7. Type "more /usr/news/mh" for details.

The file "/usr/news/mh" includes more detail about the upgrade; the key issue is that only users of the *MH* system need to be bothered with the details (and noise) that might otherwise be put into "/etc/motd." Along with the MOTD message and "/usr/news" file, we post a news article in a local news group, "purdue.cc.general;" in this example, the new article would probably have the same contents as the file "/usr/news/mh." If we have enough lead time, we also put an article in our bimonthly newsletter describing the major changes. The idea is to keep our customers well informed of changes. Unfortunately, with all this effort a few customers will still be taken by surprise; when this happens, we do our best to aid in the transition.

There is much more to keeping everyone informed than posting news articles and messages-of-the-day about up-coming system changes. With a total of more than twenty systems programmers (with *root* access) working on our UNIX systems, information about what each programmer is doing must be available to minimize our interference with each other.

Perhaps William Livingston said it best in his book, *The New Plague: Organizations in Complexity:*

"Computer departments want good documentation (no standards about what good means) supplied to them but avoid developing it themselves

[9] We highly suggest that some editing be performed. *Instck* does a good first attempt, but you will want to fine-tune some items.

[10] Most people ignore messages in the Message-Of-The-Day if the signal-to-noise ratio is low. If customers know that only **important** messages are displayed in the MOTD, then the readership goes up. This topic is worth a paper by itself.

```
Newsgroups: purdue.cc.log.unix
Subject:
Expires:
References:
Sender: kcs@houdini.cc.purdue.edu (Kevin C. Smallwood)
Reply-To: kcs@houdini.cc.purdue.edu
Followup-To: poster
Distribution: purdue
Organization: Purdue University Computing Center
Keywords:
Approved: kcs@houdini.cc.purdue.edu
Date:    Sun Aug  5 16:21:58 EST 1990

What:

Systems: VAX          (j l mace)
         Sequent      (expert mentor sage tyro)
         Sun          (staff pop element [clients])
         ETA          (boiler)

Documentation Affected:(yes no)

Changes Reported To: Berkeley DEC ETA Sequent Sun

Files:
```

Display 1 – Current template for electronic logbook as if it were poison. The culture supports each programmer to make changes continuously to vital software without writing it down. The only way to know what the system software is to never miss a day of work and establish close enough friendships with developers so that you have frequent verbal updates on the changes. Programmers soon get to be indispensable. The entire scene is out of control.''

I will address the first statement about documentation later. First I want to discuss how changes are communicated within the Group. Several years ago, information about all system changes were written in a paper log book sitting next to a system console. This method worked reasonably well. The most difficult part was developing the discipline to write all the details about a change in a form that other programmers could understand. A few years ago, we noticed that another computing facility group on campus, the Engineering Computing Network, or ECN, was posting their system log of changes and events to a local[11] news group. By reading this local news group, the Computing Center Staff was better informed about changes made by the ECN Staff. We decided to try an experiment where

[11]This news group was local to Purdue University, not just to the ECN.

```
Newsgroups: purdue.cc.log.unix
Subject: elm installed.
Reply-To: lindley@mentor.cc.purdue.edu
Followup-To: poster
Distribution: purdue
Organization: Purdue University Computing Center
Approved: lindley@mentor.cc.purdue.edu

Date:      Mon Jul 30 01:44:23 EST 1990

What:      I installed elm (version 2.3) on all PUCC machines and removed the
           test directories I set up.  I also did a lot of work on their
           Makefiles (changing things to use install, mkcat, etc).  You
           should be able to run the Configure script and get some good
           Makefiles now.  I went ahead and installed the manual pages.  I
           checked-in all the sources and I'll make a general notice
           posting about it sometime tonight.

Systems:   VAX              (j l mace)
           Sequent          (expert mentor sage tyro)
           Sun              (staff pop element [clients])

Documentation Affected:     (yes)
                            answer.1u
                            autoreply.1u
                            checkalias.1u
                            elm.1u
                            fastmail.1u
                            filter.1u
                            frm.1u
                            listalias.1u
                            messages.1u
                            newalias.1u
                            newmail.1u
                            printmail.1u
                            readmsg.1u
                            wnewmail.1u

Files:     /usr/unsup/elm2.3                          [rm -rf]
           /usr/unsup/lib/elm2.3                      [rm -rf]
           /usr/unsup/elm/*                           [OLD]
           /usr/unsup/lib/elm/*                       [OLD]
           /usr/msrc/vax/unsup/elm/Configure          [RCS]
           /usr/msrc/vax/unsup/elm/Makefile           [RCS]

           <Many lines deleted from this example>

           /usr/msrc/vax/unsup/elm/utils/printmail.sh[RCS]
           /usr/msrc/vax/unsup/elm/utils/readmsg.c   [RCS]
```

Display 2 – Example of a recent *log* message we developed a standard electronic template that described most types of system changes on the Computing Center's UNIX systems. We created a local news group called "purdue.cc.log" and a program (originally named *Plog* after Larry Wall's *Pnews* program) that allowed the invoker to edit the template and post it to the "log" news group. At first, we used the electronic log book and the paper log book in parallel; the paper log book was abandoned forever after only a few months of using the electronic log book.

The current UNIX template resembles Display 1.

The "Date:" is automatically supplied along with the network news system article header lines. The programmer is responsible for providing a meaningful "Subject:," and, optionally, a "Keywords:" line.

The "What:" information is the most important. We noticed almost immediately that more detailed information was supplied in the electronic form verses the hand-written form. Since I prefer to type over writing by hand, I can understand the reasons. The "Systems:" lines are used to show which systems (listed by host name) are affected.

The "Documentation Affected:" line reminds the programmer that documentation may need to be updated. The programmer must remove either the "yes" or "no", so this line cannot be easily ignored. If "yes" is left, the programmer would then list the affected manual pages or documents by inserting lines after the "Documentation Affected:" line.

The "Changes Reported To:" line lists the major *vendors* with UNIX based systems at the Computing Center. This is another reminder that bug fixes should be reported to the sources of the software so that we only need to fix the bug once.

The "Files:" line(s) allow the programmer to list the files that were changed. We have developed a shorthand code for showing that source files were checked-in under RCS, and that binaries were backed up to "OLD" subdirectories. See Display 2 for a recent *log* message.

All Computing Center Staff may now read the details about changes made to the UNIX systems. Furthermore, since this news group is available at all facilities with network news access at Purdue University, other departments can read about software changes made at the Computing Center. Information about bug fixes is quickly spread on campus.

After the success of the UNIX *log* system, we extended the original command (and renamed it *log)* to take an argument for the specific log you wanted to update. We now have different electronic logs and templates for engineering hardware changes, Operator activities and the different operating systems found at the Computing Center. Other departments at Purdue University have started logging their system activities using a local news group. We are now kept reasonably informed about changes happening at the Computing Center and at other departments on the Purdue Campus.

If a programmer just *install*'s a program that immediately causes problems, it is unlikely that a *log* message, which shows who installed the dysfunctional program, has been posted. The *install* command was extended to use the *syslog*(3) facility. When *root* uses *install* to install a new file, information is written to a log file. In the example above, we could quickly look at the last few lines of the "install.log" file to see who *install*'ed the broken program. This happened recently with the "/bin/login" program; it is not a good idea to have a broken "/bin/login" program installed for very long! With the *log* information, the exact program ("/bin/login") was identified, and the author was quickly contacted to fix the problem. We extended the logging facility to the *purge* command so we could track programs expired from our "OLD" subdirectories. *Purge* also uses the "install.log" file.

It is ironic that we insist on good documentation to do our work, yet programmers are notorious for being the last to produce good documentation for others. How often do we say that the source code is the ultimate documentation in the UNIX Operating System? We addressed this problem by strongly stating that major changes to production software will not be installed until the changes to the documentation (typically the manual pages) have been reviewed and approved. Exceptions exist; however, most changes can be reflected in accurate documentation. We established some guidelines for what makes a good manual page;[12] the "standards about what good means." We then set up a peer review system where manual pages are submitted and distributed to other programmers for review. Remember, we demand good documentation; we can often be the best critics of each other's manual pages. This has a secondary effect: by reading the manual page and casually testing the new command and options, other programmers become aware of new utilities or options that they never used before. Sharing of information is encouraged. After the manual page is reviewed and approved, the principal programmer *install*'s the new program and updated manual page.

Installing manual pages used to be cumbersome. The manual page for *catman*(8) says it all:

Acts oddly on nights with full moons.

People will take the path of least resistance. If it is difficult to maintain manual pages, somehow it won't get done. We developed a new program

[12]This, too, is worth a paper by itself. Of course, the topic is highly subjective.

called *mkcat*(8L) (make cat) that helps with the installation and maintenance of manual pages.

Mkcat does not insist that manual pages are written using only the *-man* macros;[13] the programmer is free to use the *-ms, -me,* or *-mm* macros. More importantly, *mkcat* uses the *install* command to install (and optionally *compress*(1)) the new *cat* version of the manual page and updates the *whatis*(1) database used by the *man*(1) command. If the manual page is set up correctly, the programmer uses *mkcat* to install the new or updated manual page. The many details are handled automatically by the *mkcat* command. By making it easier to maintain the manual page system, it is kept more consistent and up to date.

Moving Or Removing Existing Programs

Occasionally, existing programs need to be moved to another location or removed from the system entirely. We have a directory called "/usr/unsup/bin" that is reserved for *unsupported* programs at the Computing Center. "/usr/unsup/bin" is not in the system default **PATH.** What this means is that if a program in "/usr/unsup/bin" is reported as having problems, we may not be able to help. Sometimes , we don't have the expertise or manpower to help; at other times, we may not have the source code to modify. In general, we try to locate new programs in "/usr/unsup" since it is easier to delete a troublesome program from there than one installed in a directory located in the default **PATH.** If use of the new program increases, we can always provide *support* and move it to "/usr/local/bin," for example, which is in the default **PATH.**

The use of some programs declines as needs and technologies change. Sometimes it is reasonable to drop the *support level* of a program to *unsupported.* This implies that we move the binary from a directory in the default **PATH** to one not in the default **PATH** (i.e., "/usr/unsup/bin"). Posting news articles, placing messages in the MOTD, and writing articles for our newsletter are helpful, but some invocations of a program are hidden in a shell script. Shell scripts often don't read news articles or the MOTD. Our solution is to replace the real program with another (front-end) program that displays an informative message about the move and then executes the real program. This also catches the customers who miss our other efforts to publicize the move. While some shell scripts redirect the informative messages to "/dev/null," we trap most of them. At some point in the future, the real program will be moved and the old *front-end* program would be discarded.

[13]Although most manual pages are still written using the *-man* macros.

We decided that this should be automated. The *mvprog* program was developed. *Mvprog* creates a front-end program that displays a message detailing the move, new location, date it will happen (default is thirty days from installation) and a contact person in case of questions or concerns. The front-end program pauses for a few seconds and then *exec*(2)'s the real program. *Mvprog* moves the real binary (target) to its name with a dot prefix (e.g., "/bin/.ls") and installs itself as the target file (e.g., "/bin/ls"). *Mvprog* also queues an *at*(1) job that will move the real program to the location specified in the invocation of *mvprog,* and then remove the *front-end* program. The *at* job will also send the invoker of *mvprog* mail when it made the move reminding the invoker that documentation may need to be updated to reflect the new location.

Sometimes programs need to be removed from the system entirely. *Rmprog* handles this task similarly to *mvprog.* The *at* job removes both the front-end program and the real program when it is time. In the meantime, the front-end program has informed the program's users of its impending death. If the users of that program continue to need its services, then action is taken to reverse the removal.

Mvprog and *rmprog* provide a means to change the location or existence of production software while keeping the customers informed of those events. These tools also relieve the burden from the programmer or administrator by automating the tasks required to manage those future events.

Acknowledgements and Credit

Jeff Smith initially took the *install* shell script and turned it into a C program that supported backing installed files to an "OLD" subdirectory. Jeff has continued providing input into the form that *install* takes today. Jeff also reminds us of the importance of service to the customer, and where our jobs and paychecks really come from.

Kevin Braunsdorf added many extensions to *install* to support new options and the *install.cf* configuration file. Kevin created the *instck* program to help us maintain what we call a *pristine* system. Kevin also developed the *mkcat* replacement for *catman*(8) that has made our manual pages easier to manage and keep current and consistent. Don Ault and others have worked long hours to edit the manual pages so they are consistent.

Rich Kulawiec developed the *log* system we use to keep each other informed. Dave Stevens wrote the *tickle* shell script that keeps us honest and the first *purge* shell script which Kevin Braunsdorf has since turned into a C program. Matthew Bradburn and Chris Daniels developed the current versions of *rmprog* and *mvprog.*

Finally, much of the **real** credit for the guidelines and tools described in this paper goes to the UNIX Systems Programming Group as a whole. So many things have evolved and continue to evolve as a group effort.

After Thoughts

Most of the guidelines reported above are not *earth-shattering.* Hopefully, you will see this as a "blinding flash of the obvious." It is all based on common sense and some experience. We do not purport to be **the** authority on systems management. We hope that the reader will get some ideas from the guidelines and tools we described in this paper and develop his own based on his computing environment and priorities. All the policies, procedures, guidelines or tools in the world will not help you if you have not established a mind-set that you and your team want to provide a system that is responsive to the customer. If you and your organization really care about being responsive to customer needs, then well over fifty percent of the work is done. We are continually evolving our guidelines and tools to fit the new demands. We welcome comments and suggestions from others faced with similar problems.

The availability of the many tools described in this paper should be made possible by the "comp.sources.unix" news group. If you have not seen them yet, they should be available shortly.

References

[1] Livingston, William L. *The New Plague: Organizations in Complexity.* New York: F. E. S. Limited Publishing, 1986.

[2] *UNIX Programmer's Reference Manual, 4.3 Berkeley Software Distribution.* Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, April 1986.

[3] *UNIX System Manager's Manual, 4.3 Berkeley Software Distribution.* Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, April 1986.

[4] *UNIX User's Reference Manual, 4.3 Berkeley Software Distribution.* Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, April 1986.

[5] *UNIX User's Supplementary Documents, 4.3 Berkeley Software Distribution.* Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, April 1986.

Kevin Smallwood is the Manager of UNIX Systems Programming for the Purdue University Computing Center, Math Science Building, West Lafayette, Indiana 47907. Kevin can be reached at (317) 494-1787, or electronically at "kcs@cc.purdue.edu." Kevin graduated in 1979 with a BS in Computer Science from Purdue University. Kevin worked for the Radar Systems Group of the Hughes Aircraft Company, Technical Systems Consultants, Inc., the Department of Computer Science at Purdue University, and finally the Computing Center at Purdue University. Areas of interest include performance tuning, file systems and service to the customer.

NAME

install – update files or directories in a controlled environment

SYNOPSIS

install [**–1Dclnpqsv**] [**–C***checklist*] [**–H***hardlinks*] [**–S***symlinks*] [**–g***group*] [**–m***mode*] [**–o***owner*] *files destination*

install –d [**–hnqrv**] [**–C***checklist*] [**–g***group*] [**–m***mode*] [**–o***owner*] *directory*

install –[**hV**] [**–C***checklist*]

install –R [**–1dlnqsv**] [**–C***checklist*] [**–H***hardlinks*] [**–S***symlinks*] [**–g***group*] [**–m***mode*] [**–o***owner*] *destination*

DESCRIPTION

Install is a tool for updating system files in a controlled environment. The design philosophy of *install* is to automate the installation process and put it as much out of reach of human carelessness as possible, while providing a flexible, easy to use tool. *Install* provides the following features:

- *Install* increases system security by providing a controlled installation environment. It checks the actions of the superuser against a configuration file that you build (either by hand or using *instck*(1L)), and can prevent grievous mistakes such as installing a shell setuid to a privileged user, or installing a world-writable password file. An appropriate configuration file can guarantee that files are installed with correct owner, group and mode, that *strip*(1) is run on binaries and *ranlib*(1) is run on libraries. Regardless of whether you create a configuration file, *install* warns you of many possible errors, unless you make it quiet with its **–q** option. For instance, if you install a new version of the *ex*(1) editor and forget to update its links, *install* will notice the extra links to the existing version and warn you that you might be making a mistake.
- Installed files do not overwrite current versions. The current version is backed up to a subdirectory of its dot directory (named "OLD" by default), where it may be easily re-installed in the case of an unforeseen bug. The companion program *purge*(1L) removes these backed-up files after a user-specified interval. By default, if you repeatedly install new versions of a file, *install* creates a series of backups, providing a primitive form of version control.
- *Install* increases accountability by logging the actions of the superuser. This makes it easier to track down errors after the fact.
- *Install* simplifies Makefiles by allowing you to combine operations that would require several steps into a single one (e.g., you can specify in a single command line a file's ownership, group, mode, whether to run *strip*(1) or *ranlib*(1), and which hard or soft links should be made).
- Despite its power and potential complexity, *install* is easy to use interactively because it intuits appropriate installation parameters for you, either by using those associated with an existing file, or its compilation-dependent defaults. In most cases you do not have to specify a file's owner, group, or mode unless you want them to be different from an existing file or the compilation-dependent defaults. Typical interactive usage is simply "install file destination."
- *Install* is as careful as it can be to complete an installation once it is begun. There is a point in the code where *unlink*(2) and *rename*(2) must be executed in close succession, and that is the only window in which a system crash or a signal might leave system files in an inconsistent state (unfortunately, this is not true under operating systems that do not provide an atomic

rename(2) system call). This is true even when *install* must copy files across file system boundaries.

- *Install* may also be used to remove (de-install) files in a controlled manner.
- Finally, *install* currently runs on a variety of architectures and operating systems and is easy to port to new platforms.

USAGE

Terminology

The user specifies one or more *files* to install, and a *destination*, which may be either a full or relative pathname ending in a file name or an existing directory name (if the directory does not exist, *install* thinks you mean to create a new file). The special name "–" may be used for the *file* argument to indicate stdin (see EXAMPLES). In this case, *destination* **must not** be a directory, since *install* cannot guess the name the file should have when installed.

Because the user may specify more than one *files*, and *destination* may be an existing directory or a file which may or may not exist, *install* must also internally keep track of a *destdir* ("destination directory", i.e., the directory in which to place the file to install), and a *target* (the full pathname that each file to install will have when it is installed in *destdir*). The *target* and *destdir* are constructed from the *files* and *destination* arguments as described below.

For each name in *files*, *install* determines a *target* name as follows: If *destination* is an existing directory, *install* catenates the last component of *file* to *destination* to arrive at the *target* name. If *destination* does not exist or is an existing file, *install* takes *destination* to be the *target*. In the latter case *destdir* is simply *destination* minus its last component. If this reduction leaves *destdir* empty then it is set to ".". (E.g., if *destination* were */etc/motd* then *destdir* would be */etc*, but if *destination* were just *motd* then *destdir* would be ".".) N.B.: If more than one *files* are specified, *destination* **must** be an existing directory.

Examples are the easiest way to clarify this terminology.

In the command "install motd /etc/motd":

file:	motd
destination:	/etc/motd
destdir:	/etc
target:	/etc/motd

In the command "install motd /etc":

file:	motd
destination:	/etc
destdir:	/etc
target:	/etc/motd

In the command sequence "cd /etc; install motd.new motd":

file:	motd.new
destination:	motd

```
destdir:     . (dot)
target:      ./motd
```

Installation Parameters

If the file permissions, ownership or group ownership are not specified on the command line and *target* exists, *install* duplicates its group, ownership, and mode. Otherwise, if the *target* doesn't exist and the invoker is the superuser, *install* uses its compilation-dependent defaults. Otherwise, *install* uses the effective uid, the effective gid, and a compilation-dependent mode. *Install* may also be configured to inherit the mode and ownerships from the *destdir*. (Use the **–V** option to view the compilation-dependent defaults.)

Note: *install* can only change ownership if invoked by the superuser; however, any user may specify a different group as long as the group is allowed by *chgrp*(1).

Method of Operation

Install first checks the proposed owner, group, and mode against the configuration file. It also checks whether the *target* should have *strip*(1) or *ranlib*(1) run on it after installation. *Install* aborts if it finds discrepancies between the configuration file and the proposed installation parameters. (If necessary, you can override the configuration file with "–C */dev/null*.")

Install then looks for an existing *target* and backs it up to a subdirectory of *destdir* named "OLD", which *install* will create if it doesn't exist. The backup is actually just a hard link to the existing *target*. (If a backup file of the same name already exists in "OLD", *install* first renames it by appending its process id). For security reasons, *install* drops setuid and setgid bits when files are moved to the "OLD" directory. After backing up an existing *target*, *install* temporarily moves *file* to *destdir/OLD/random-name*. This step is taken to ensure that both *file* and *target* are in the same file system so that *rename*(2) may be used for the final installation. This reduces the window in which files might be left in an inconsistent state due to a system crash or signal. Consequently the "OLD" subdirectory must not be a file system mount point, since the *rename*(2) would fail.

Install then unlinks the existing *target* (leaving the backup) and renames *destdir/OLD/random-name* to *target*.

Next *install* updates any hard links and soft (symbolic) links given under the **–H** or **–S** options. All links point at the installed *target*. Existing symbolic links which point to the correct file are left unchanged, otherwise removed and replaced with the correct spelling. Existing correct hard links are unlinked and replaced, without a backup. Links which point to a file other than the *target* are backed up to "OLD" and linked. *Install* prints a warning in this case.

OPTIONS

–1 After a successful installation *install* removes any previous backup in "OLD". Thus *install* will keep exactly one previous revision of the installed file.

–c Do not unlink the *files*.

–C*checkfile*
Search *checkfile* for an expression that matches the *target*. If *install* finds such an expression it will check the proposed modes (etc.) against those listed in the checkfile; any differences cause *install* to abort the installation. This mechanism is provided to protect the superuser from installing, for instance, the shell setuid root, as in:

```
install –m7555 –o root –g wheel sh /bin
```

(note the extra "5"). See *install.cf*(5L) and *instck*(1L).

-d Build a directory rather than a plain file. If the directory is an OLD directory it is built with appropriate modes (see -V below).

-D Don't back up an existing *target* (the "Destroy" option). This is useful when a trivially correctable problem such as a spelling error in a print statement is found in a recently installed product, or when the *target* can be regenerated easily and is installed frequently. Sites that do not wish to keep backups but still want to take advantage of the checkfile could set this option in the environment.

-g*group*
Install the file with group ownership *group*. If no **-g** option is given *install* will decide the group to use:
- if there is an existing *target* use its group
- if we are the superuser use a compilation-dependent group
- else use the effective group id

-h Print a summary of *install*'s usage (the "help" option).

-H*hardlinks*
Specify a colon separated list of hard links that should be made to the *target* after it is installed. (See EXAMPLES below.)

-l Run *ranlib(1)* on the installed *targets*. Under System V this option has no significance, but **must** be given to pass the default checkfile (see *install.cf*(5L)). This allows Makefiles to work under all versions of UNIX.

-m*mode*
Install the file with permissions *mode*. *Mode* may be given in either octal mode or in the symbolic form used by *ls*(1) (e.g., "755") is equivalent to "rwxr-xr-x"). If the **-m** option is not given, *install* will decide the mode to use:
- if there is an existing *target* use its mode
- if we are the superuser use a compilation-dependent mode
- else use a compilation-dependent user mode

-n Give an approximate execution trace by printing the (almost) equivalent shell commands, but don't do anything. This is useful for debugging *install* or seeing what a difficult command line would do if you ran it.

-o*owner*
Change ownership of *file* to *owner* (superuser only). If no **-o** option is given *install* will decide the owner to use:
- if there is an existing *target* use its owner
- if we are the superuser use a compilation-dependent owner
- else use our effective uid

-p Preserve the time stamp of *files* in *targets*.

-q Normally *install* informs you about a variety of possible errors. This option turns off that behavior and is not recommended except for special circumstances. Caveat emptor.

-r Under **-d** build all intervening directories between "/" and *destination.*

-R Remove (de-install) a file by moving it into the OLD subdirectory. A temporary shell script is created to replace the *target*, installed (which removes the existing *target* to "OLD"), and removed.

–s Run *strip*(1) on the installed *targets*.

–S*symlinks*
Specify a colon separated list of symbolic links that *install* should point at the installed file. (See EXAMPLES below.)

–v Be verbose. Run *ls*(1) on the backed up file and the *target*. Notify the user of all side effects of this installation.

–V View *install*'s version and compilation-dependent owner, group and mode tables.

EXAMPLES

install motd /etc
Install *motd* as */etc/motd*. If */etc/motd* exists move it to */etc/OLD/motd* and duplicate its ownership, group and mode; otherwise, use defaults. Create the directory */etc/OLD* if it does not exist.

install –c1 –m 755 foo.sh /etc/foo
Install *foo.sh*. as */etc/foo*. Do not unlink *foo.sh* after the installation. Set permissions appropriate for a shell script on */etc/foo*. If */etc/OLD/foo* exists, overwrite it instead of moving it to a new name (i.e., retain a single backup of */etc/foo* in */etc/OLD*).

install foo bar baz /usr/lib
Install the files *foo*, *bar* and *baz* as */usr/lib/foo*, */usr/lib/bar*, and */usr/lib/baz*. Use the modes of the existing files or defaults.

install –vsm6751 –oroot –gkmem sendmail /usr/lib
Install *sendmail* as */usr/lib/sendmail*, owned by root, grouped to kmem, and with the setuid and setgid permission bits set. Strip */usr/lib/sendmail* after its installation and be verbose.

install –d –m –rwxrwxrwt /tmp
Build the directory */tmp* with the default owner and group, mode 777, and with the "sticky" bit set.

install –c –m1755 –Hview:vi:edit:e:/usr/bin/ex a.out /usr/ucb/ex
Install the *ex* editor with all of its hard links (*/usr/ucb/view*, */usr/ucb/vi*, */usr/ucb/edit*, */usr/ucb/e*, and*/usr/bin/ex*). Replacing **–H** with **–S** would cause *install* to build symbolic links on machines which support them.

install –d –r /usr/local/lib/mk
Recursively build any and all of the directories */usr*, */usr/local*, */usr/local/lib*, and */usr/local/lib/mk* that do not already exist. Silently do nothing if they already exist (useful in Makefiles).

install –V
Output the version of install and a table of compiled in defaults. Output when run as the superuser might look similar to this, depending on the compilation defaults:

```
install: version: $Id: main.c,v 6.7 64/02/15 16:21:41 ksb Exp $
install: configuration file: /usr/local/etc/install.cf
install: syslog facility: 144
install: superuser defaults:
install: owner is file=root        dir=root        OLD=root
```

```
install: group is file=binary     dir=binary     OLD=binary
install: mode is  file=0755       dir=0755       OLD=inherited
```

rsh some.other.host install – /etc/motd < motd.some.other.host

This example shows a way to use ‘‘–’’ to advantage. It is often useful when *rdist*(1) is overkill or otherwise inappropriate. For instance, if you had a Makefile that generated files named *host1.motd*, *host2.motd*, *host3.motd*, etc., and wanted to install them on those hosts, you could do something like this:

```
for host in host1 host2 host3; do
        rsh $host install – /etc/motd < $host.motd
done
```

DIAGNOSTICS

Unless made quiet by –q, *install* will warn the installer if:

- the owner, group, or mode changes
- the setuid, setgid, or sticky bits change
- a setuid program is loaded with the ‘‘#!’’ magic number
- *target* does not exist (this is a prophylactic against typographical errors—see CAVEATS below)

Install will abort the installation if:

- *install* cannot make a backup of an existing *target*
- a setuid program´s owner was not specified with the mode
- a setgid program´s group was not specified with the mode
- the specified owner, group, or mode failed to match the checkfile
- the superuser installs a setuid program that is not in the checkfile (this is a compile time option)

ENVIRONMENT

The environment variable **INSTALL** may be used to set command line options. Such options are read before any explicit command line options, e.g.

```
INSTALL=-v ; export INSTALL
```

will turn on ‘‘verbose’’ mode for all subsequent invocations of *install*.

BUGS

Install does not use file locking, so it’s possible for two competing *install* processes to lose data.

The trace option (–n) doesn’t always show exactly what *install* would do. It will show OLD directories being made several times, and inherited modes don’t propagate correctly under **–drn**.

CAVEATS

Install will not build character special files. Use *mknod*(8) instead.

If /bin doesn’t exist, the command:

```
install ls /bin
```

will make */bin* be a copy of the binary *ls*. This is an unavoidable consequence of allowing the *destination* to be a directory. You can avoid this by using ‘‘install –d *directory*’’ in Makefiles to ensure that destination directories exist, e.g.,

```
install: product
        install -d /bin
        install product /bin
```

Install does nothing and exits normally if the directory already exists, so it's safe to include lines like this in your Makefiles. You can also use *instck*(1L) to ensure that all system directories are built correctly.

Install can cover up mistakes of omission in Makefiles by copying the modes on a previously installed target. For example, the invocation of *install* in a Makefile might not specify that the *destination* should be setuid root, but as long as the *target* existed *install* would hide the error by copying the modes of the existing *target*. The problem in the Makefile would not be found unless the *target* were removed prior to installation. Use *instck*(1L) to generate a configuration file to avoid this problem.

Install and *purge*(1L) assume that they own the file namespace in the ''OLD'' subdirectories. If other programs create, modify or delete files in the ''OLD'' subdirectories, they will probably collide with one of *install* or *purge*(1L) eventually.

FILES

/usr/local/lib/install.cf	the default check list file

AUTHORS

Kevin Braunsdorf, Purdue University Computing Center (ksb@cc.purdue.edu)
Jeff Smith, Purdue University Computing Center (jsmith@cc.purdue.edu)

SEE ALSO

chgrp(1), instck(1L), ls(1), make(1), ranlib(1), strip(1), intro(2), syslog(3), install.cf(5L), chown(8), mknod(8), purge(8L)

NAME

install.cf – a check list to confirm correct installations

DESCRIPTION

Install(1L) allows the system administrator to update important files while maintaining a backup copy of any old versions. *Install* checks the actions of the superuser against a list of important files, to prevent careless security breaches. The –C option controls which checklist *install* examines.

Install searches *install.cf* for an *sh*(1) *glob* expression that matches the file to be installed. When it finds a matching expression it checks the proposed owner, group, and mode against those listed in the checklist; any variation causes *install* to abort. This mechanism is provided to protect the superuser from installing, for instance, the shell setuid:

```
install -m7555 -o root -g wheel sh /bin
```

(note the extra `5´).

Each data line in *install.cf* consists of 6 fields;
- a glob expression
- a mode, in either octal or symbolic format
- a symbolic owner
- a symbolic group
- a *strip*(1) or *ranlib*(1) indicator
- an optional comment

Blank lines and lines which begin with a pound sign (#) are ignored.

If the glob expression begins with a slash (''/'') it is matched against the full *target*, otherwise it is matched against only the last component of the *target*. (For the definition of *target* see *install*(1L).) For instance, the expression

```
*.a
```

will match every library (file ending in `.a´), while

```
/lib/*.a
```

will only match libraries in ''/lib''.

The mode field may be filled with either an octal number (discouraged) or a symbolic mode (see *ls*(1)). Any bit specified as on **must** be on, any bit specified as off (–) **must** be off. The symbolic mode may also include `?´ to indicate an optional bit. For instance the mode

```
–r?–r—r—
```

will match either –rw–r—r— (0644) or –r—r—r— (0444). If a setuid, setgid, or sticky bit is set without the underlying execute bit being set the `s´ or `t´ should be given in uppercase.

Some combinations of modes are not expressible in the above format, e.g. 0755 with an optional setgid bit. For such modes a special format is allowed:

mode/optional-mode

in which case all the bits in *optional-mode* are taken as optional. To express our example mode one might use:

```
drwxr-xr-x/02000
```

(this is a good mode for a directory under SunOS).

The type bit in the symbolic mode may be any of the *find*(1) type bits, or an exclamation point (''!''). The ! is used to indicate files that should **never** exist, and may prefix an auxilary file type.

The owner (group) column may either contain an asterisk (‘‘*’’) or an alphanumeric identifier. If an asterisk is given any valid owner (group) may be used. No numeric user (group) identifiers are allowed. A owner (group) name may be prefixed by an exclamation point (‘‘!’’) which indicates that only this owner (group) is not allowed.

The strip/ranlib indicator follows this table

n	do not run either *strip*(1) or *ranlib*(1)
l	run *ranlib*(1)
s	run *strip*(1)
*	allows either to be run, requires neither

It makes no sense (to the authors) to run both *strip*(1) and *ranlib*(1).

Install displays the comment section when the file is installed.

A double quote ('') substitutes the entry from the previous line. An equals sign (=) substitutes *install*'s default value for a field.

The tool *instck*(1L) may be used to generate configuration files for named files (assuming the current ones are correct).

EXAMPLE

```
# this file tells install(1L) what files to check as special
# file          modes         owner    group   strip   comment
# name                                         ranlib  for user
/unix           –r??r—r—      binary   *       n       new kernel
/vmunix         ''            ''       ''      ''      ''
/dynix          ''            ''       ''      ''      ''

/bin/sh         –rwx?–x?–x    *        *       *       do not setuid
/bin/su         –rws?–x?–x    root     *       *       must setuid, not group
/bin/init       –rwx———       root     *       *       no world execute
/etc/init       ''            ''       *       *       no world execute
/etc/passwd     –r?–r—r—      root     *       n       forbid world write
/etc/group      ''            ''       *       n       ''
/tmp            drwxrwxrwt    binary   *       n       must be sticky at PUCC
/usr/tmp        ''            ''       *       n       sticky
/usr/ucb/eyacc  –rwxr–x?–x    *        *       n       no strip
/usr/ucb/lisp   ''            ''       ''      ''      ''

core            !-??-?--?--   *        *       n       a bogus core file
```

AUTHORS

Jeff Smith, Purdue University Computing Center (jsmith@cc.purdue.edu)
Kevin Braunsdorf, Purdue University Computing Center (ksb@cc.purdue.edu)

SEE ALSO

install(1L), instck(1L), ls(1), sh(1), strip(1), ranlib(1)

NAME

instck – look for improperly installed files

SYNOPSIS

instck [–dilSv] [–C*checklist***] [–g***group***] [–m***mode***] [–o***owner***]** *directories*
instck [–G] [–dl] *directories*
instck [–hV]

DESCRIPTION

Instck reads a system *checklist* for a list of correct modes for installed files. It scans the file systems for incorrect owners, groups, modes etc.

For instance, if the shell were installed setuid root via:

```
install –m7555 –o root –g wheel sh /bin
```

(note the extra `5´) instck might find it.

If the given *directories* end with ''OLD'' they are only searched for half installed files and insecure backup modes. Patterns read from the *checklist* which do not start with a slash ('/') are examined from the remaining *directories*.

Instck prompts stdin with suggested repairs under the **–i** option.

OPTIONS

–C*checklist*
Search *checklist* for the expressions to match the installed files (see *install.cf*(5L).

–d Do not check the contents of a directory, as in *ls*(1).

–g*group*
Set the default group for installed files.

–G This option will generate a checklist file for the given directories. Such a file may have to be edited to be useful.

–h Print a summary of *instck*'s usage.

–i Prompt the user with suggested shell commands, for example if /tmp was the wrong mode *instck* might prompt with:

```
instck: '/tmp' mode 0777 doesn't have bits to match drwxrwxrwt (1777)
instck: chmod 1777 /tmp [nfhqy]
```

A reply of 'y' or 'Y' will run the proposed command, 'f' will skip to the next file.

–l A long list will be made of all the files that didn't match any rule. Under **–G** all files will be put in the checklist.

–m*mode*
Set the default mode for installed files.

–o*owner*
Set the default owner for installed files.

–S Run as if geteuid() returned zero.

–v Be more verbose.

–V Give version information.

EXAMPLES

instck /bin /usr/bin /usr/ucb /usr/local/bin /etc
: report inconsistent installations in the listed directories.

instck –v /etc
: report inconsistent installations in /etc, be very verbose.

instck –G /bin
: generate a checklist for /bin and all the files in /bin.

instck –V
: Report version information, e.g.

```
instck: version: $Id: main.c,v 6.0 90/03/08 15:34:49 ksb Exp $
instck: configuration file: /usr/local/etc/install.cf
instck: defaults: owner=root    group=binary    mode=–rwxr–xr–x
```

FILES

/usr/local/lib/instck.cf   the default *checklist* file

AUTHOR

Kevin Braunsdorf, Purdue University Computing Center (ksb@cc.purdue.edu)

SEE ALSO

ls(1), chown(8), chgrp(1), chmod(1), install(1), ranlib(1), strip(1), geteuid(2), syslog(3), install.cf(5L)

NAME

mvprog – move a command and inform users

SYNOPSIS

mvprog [**-hnv**][**-t** *when*][**-u** *who*][**-w** *why*] *from to*

or

mvprog [**-hnv**][**-t** *when*][**-u** *who*][**-w** *why*] *files directory*

DESCRIPTION

A system administrator should use *mvprog* to move a command from one directory to another. After moving *from* to *to*, it creates a program which it places in *from* that, when invoked, prints out a message about the new location of the command and prompts the invoker to hit **return** (if the command is run from a tty). Once return is pressed, this command executes the command in the *to* location. *Mvprog* also queues an *at*(1) job which will remove the *from* command in *when* (30, by default) days. System administrators must change the documentation and the makefile for the source, if necessary. The *at* job sends mail to the invoker as a reminder.

OPTIONS

–h	Print a short help message.
–n	Don't really do anything.
–v	Produce verbose output.
–t*when*	Arrange for the link in the old directory to go away in *when* days.
–u*who*	Print *who* as the source to conctact about details concerning the command move.
–w*why*	Print the string *why* when the command is executed, in addition to the information that is normally printed.

EXAMPLES

We want to move */bin/foo* to */usr/bin/foo*:

```
# mvprog /bin/foo /usr/bin
```

Afterward,

```
$ /bin/foo
```

will print:

```
This program, /bin/foo, has moved to /usr/bin/foo.
You will no longer be able to invoke it via this path after <date>.
Send questions and comments regarding this move to "<user>".
```

After printing this message, the */bin/foo* invokes the original command, now in */usr/bin/foo.*

BUGS

The program put in the old location may fail to invoke the real program if it is a shell script without a ''loader card,'' even though a shell would be able to execute it directly.

FILES

/tmp/pmXXXXXXX/*
temporary directory for compilation of the replacement command

AUTHOR

Matthew Bradburn, Purdue University Computing Center
Chris Daniels, Purdue University Computing Center

SEE ALSO

at(1), atq(1), atrm(1), rmprog(8L)

NAME

purge – remove outdated backup copies of installed programs

SYNOPSIS

/usr/local/etc/purge [**–ASVhnv**] [**–d** *days*] [**–u** *user*] [*dirs*]

DESCRIPTION

Purge removes backup files that are more than *days* old from OLD directories created by *install*(1L). *Purge* locates the OLD directories by recursively descending *dirs* selecting directories named "OLD" that the invoker owns.

Purge does not remove backup files with multiple hard links if it cannot find all the links. This is an indication that a product was installed without regard for multiple names (*install* warns the installer in such cases, but the error message may have been inadvertently ignored). *Purge* directs the maintainer to run *instck*(1L) to correct any such problems.

OPTIONS

–A	Allow OLD directories owned by any user to be searched for out of date backup files.
–d*days*	Days to keep backup files (default 14).
–h	Print a help message.
–n	Do not really execute commands, output what would be done (see **–v**).
–S	Run as if we are the superuser. In combination with **–n** this allows a system administrator to see what *purge* would do when run as root.
–u*user*	OLD directories may (also) be owned by this user.
–v	Be verbose by showing approximate actions as shell commands.
–V	Show version information.

EXAMPLES

```
purge -v -d7 $HOME/bin
```
Remove any backup files from my bin which are older than one week.

```
purge -Sn /bin/OLD
```
Show which files purge would remove from /bin's OLD directory.

```
purge -u news -u uucp -u adm -u lp /
```
When run as root, purge all system OLD directories on this machine.

```
purge -V
```
Output *purge*'s version information, eg.:

```
purge: $Id: purge.c,v 2.0 90/06/24 21:28:27 ksb Exp $
purge: default superuser login name: root
purge: backup directory name: OLD
purge: valid backup directory owner: root
```

BUGS

Purge does not cross nfs mount points.

This version of *purge* is slightly incompatible with the shell script.

AUTHOR

David L Stevens, Purdue UNIX Group, dls@cc.purdue.edu
Kevin Braunsdorf, Purdue UNIX Group, ksb@cc.purdue.edu

SEE ALSO

install(1L), instck(1L), rm(1), sh(1)

NAME

rmprog – inform users that a command is scheduled to go away

SYNOPSIS

/usr/local/etc/rmprog [**-hnv**][**-t** *when*][**-u** *who*][**-w** *why*] *file(s)*

DESCRIPTION

Rmprog moves each *file* to *.file* and puts in its place a program which prints a message about the command being scheduled for deletion, prompts the invoker to press **return** (if the command is invoked from a tty), and then invokes *.file*. It also submits an *at*(1) job which will remove both *file* and *.file* in *when* days (default 30). The *at* job also reminds the invoker to remove the program's source, manual pages, and *whatis*(1) entry via *mail*(1).

OPTIONS

–h	Print a short help message.
–n	Don't really do anything.
–v	Produce verbose output.
–t*when*	Arrange for the command to go away in *when* days.
–u*who*	Print *who* as the person to contact regarding the removal.
–w*why*	Print *why* when the command is executed, in addition to the information that is normally printed.

EXAMPLES

For example, we want */bin/foo* to go away in a month, because it has been outdated by */bin/bar*:

```
# rmprog -w "Use /bin/bar instead" /bin/foo
```

Afterward

```
$ /bin/foo
```

will print:

```
This program, /bin/foo, will be deleted <date inserted here>.
Use /bin/bar instead.
Send questions and comments concerning this deletion to <who>.
```

After printing this message, */bin/foo* invokes the original command, now named */bin/.foo.*

FILES

/tmp/pgXXXXXXX/*
temporary directory for compilation of replacement command

AUTHOR

Matthew Bradburn, Purdue University Computing Center

SEE ALSO

at(1), mail(1), whatis(1), mvprog(8L)

LISA IV
October 17-19, 1990
Colo. Springs, Colorado

Keeping Up With the Manual System

Kevin Braunsdorf – Purdue University Computing Center

ABSTRACT

While the manual page and "whatis" databases are complex and dynamic, the tools provided to maintain them may be insufficient if a site takes its documentation work seriously. This paper describes an integrated replacement for *catman*(8) and *makewhatis*(8) that carefully updates the manual page and the "whatis" databases in parallel. The *mkcat*(8L) (pronounced "make-cat") command integrates the features of *catman*(8) and *makewhatis*(8) and provides additional functionality that eases maintenance of the manual system and makes it safer. *Mkcat* is a drop-in replacement and requires no changes in the basic structure of the manual system database or the user agents used to extract information from it (*apropos*(1)[1], *man*(1), and *whatis*(1)[2]).

Introduction

Sites may wish to modify their manual system's database for many reasons: they may make local modifications to standard programs, develop custom software, install software from other sources (e.g., the "net"), or even remove programs. Sites that take their documentation work as seriously as their programming efforts will wish to update the supporting documentation. At the Purdue University Computing Center (PUCC) we found that the standard tools provided for maintenance of the manual system's database are insufficient to our needs. We developed the *mkcat*(1L) command to replace the standard tools *catman*(8) and *makewhatis*(8).

All references in this paper apply to 4.3 BSD-derived UNIX systems though they should apply equally well to System V UNIX systems. See the section on *mkcat* for suggestions for using *mkcat* under System V.

The BSD UNIX Manual Page System

User Agents

The *man*(1) command displays a manual page as requested by the user, using the pager specified in the environment variable "PAGER." If no formatted manual page is found in the cat*n* directories some *man*(1) commands will attempt to format the corresponding man*n* page on the fly. The *man* command has no special privileges and it need not run setuid or setgid. (However, on some systems *man* is set to run setgid to something that can write in */usr/man*. It can then install formatted pages if a cat page is out of date with respect to its source.)

The *whatis*(1) command is a link to the *man* command that performs a *grep*(1)-like search of the "whatis" database for a command. The output text is a list of single line descriptions of the matched commands. It greps for its argument on the left-hand side of the "whatis" database and prints lines that match.

The *apropos*(1) command is a link to the *man* command that performs a search of the "whatis" database for any command whose description contains a given keyword. The output text is a list of single line descriptions of the matched commands. The *apropopos* command greps for its argument on the right-hand side of the "whatis" database and prints lines that match.

The manual pages

The manual system's database comprises the formatted pages displayed by *man*, and the "whatis" database displayed by the *apropos* and *whatis* commands (or by the **–k** and **–f** *man* options). These should be maintained in parallel, since they are simply different ways of accessing the same data. In other words, if you can display a page via *man*, you should be able to access its *whatis* and *apropos* entries. Conversely, if a page is removed, the "whatis" database entry should also be removed.

On BSD UNIX derived systems the database stored under */usr/man* contains numbered directories corresponding to sections 1 to 8 of the UNIX programmer's manual. At most sites the *nroff*(1) sources to the manual pages are kept in the "man*n*" directories. (However, at PUCC we have found that by keeping the manual page source with the program source, maintainers are more likely to update the manual page at the same time as the program.) It is possible to keep only the *nroff*(1) sources on-line and format them on the fly, but most sites also keep pre-formatted versions of the pages in the numbered "cat*n*"[3] directories.

[1] *man* **–f**
[2] *man* **–k**

[3] "cat" is a historical reference to the C/A/T typesetter for which the original *troff*(1) prepared output.

It may be desirable to compress the pages in the cat directories to save disk space, at the expense of additional processing when uncompressing them for display.

It should be possible to refer to manual pages by any of the names mentioned in the "NAME" section of the page (e.g., "man printf" and "man sprintf" should produce the same result). This can be done by producing multiple versions of the cat pages, but since the pages are identical either hard or symbolic links may be used to save disk space. Note that the names for some manual pages do not represent any single product on the machine. For instance, the manual page *bstring*(3) describes the BSD C library byte manipulation functions, but there is no function named *bstring*. It should be possible to access these pages either by the generic name ("bstring") or any of the names in the "NAME" section of the page.

The whatis database

The ASCII file *whatis* contains a list of the pages in the manual with a single line synopsis describing each one. The synopsis is extracted from the "NAME" section of each page. For example, the "NAME" line from the *printf*(3) manual page (see Figure 1 below)
should produce the three lines shown in Figure 2 in the "whatis" database.

Even if the name of the manual page is not one of the names listed in the NAME section (e.g., *bstring*(3)) it should also appear in the "whatis" database.

Problems With the Standard Tools

Synchronization and Timely Updates

At PUCC the principle problem we experienced was that the "whatis" database would get out of sync with the manual page database. The consistency of the two databases depended on manual editing of the file */usr/man/whatis* when pages were installed, modified, or deleted, and people would make mistakes. The only solution with the standard tools was to regenerate the entire "whatis" database when we noticed the inconsistency, a time-consuming procedure. The standard advice is to run both *catman* and *makewhatis* each night out of *cron*(8), a procedure that has been likened to a nightly whacking of the manual system databases with a large hammer. The root problem is that the standard tools act on the entire database and cannot selectively update parts of it. At most sites the greatest portion of the database remains the same from day to day, and it's a waste of time to check it all to see if it needs updated. Further, the cat page and the "whatis" database should be updated when a manual page is installed, modified, or deleted – not later that night. However, the computing resources consumed by *catman* and *makewhatis* militates against interactive use to update a single page (each time *catman* runs it compares the timestamp of every file in the man*n* directories with the corresponding file in the cat*n* directories). These problems of synchronization and timely updates can be avoided entirely if the tool that updates the manual pages can act on selected parts of the manual page database and updates the "whatis" database at the same time.

Alternate manual databases

Some sites have directories *mann* and *mano* that contain an alternate manual system for new and old manual pages. At PUCC, we maintain all the X11R4 manual pages under the subtree */usr/man/Xman*. These separate manual hierarchies should have their own cat*n* and man*n* subdirectories and "whatis" databases. By changing the environment variable "MANPATH", users can access pages and "whatis" data under these alternate hierarchies. The standard maintenance tools should be able to cope with these alternate hierarchies, and more recent versions of *catman* can, at the expense of one invocation for each alternate hierarchy. The *makewhatis* program does not have any concept of an alternative manual hierarchy.

Text Formatting Directives

The standard BSD system provides no facility for embedding text formatting information in the manual page sources. System V Release 3 provides a primitive facility in the form of a comment in the manual page source:

.\" *keys*

which may be embedded as the first line of the manual source file. The *keys* argument could be any of "t", "e", "p" and would cause *tbl*(1), *eqn*(1), or *pic*(1) respectively to be run before *nroff*. This system is better than nothing but is still limited.

```
        printf, fprintf, sprintf — formatted output conversion
```

Figure 1 – the "NAME " line from *printf*(3s)

```
        fprintf(3s) — formatted output conversion
        printf(3s) — formatted output conversion
        sprintf(3s) — formatted output conversion
```

Figure 2 – Expected output from the "whatis" database

The *catman* program does not provide a facility for embedding formatting information. Instead it runs each manual page through *tbl* whether it needs it or not. It does not allow use of text formatters other than *nroff*, and it assumes that all manual pages are written using the *man*(7) macros. Ideally, a maintenance tool would allow the use of processors other than *nroff* and the use of any macro package. It would extract the formatting information from the manual page source itself only falling back on built-in rules as a last resort.

The Solution: *mkcat*(1L)

PUCC developed *mkcat* to address these problems. *Mkcat* takes a manual page source and formats it according to built-in defaults, global defaults specified in a ".mkcat–opts" file in the root of the manual subtree, or from embedded directives in the manual page. The formatted result is installed in the cat part of the manual subtree after backing up the existing page. Hard or soft links to the other names by which the manual page are known are made at this time. If desired, the formatted page is then compressed. Finally, the "whatis" database is updated. The entire process takes a short time and uses few computing resources. The behavior of *catman* can be simulated with a *find*(1) invocation if desired, and there is an option to rebuild the "whatis" database *in toto* if desired. The workings of *mkcat* are explained in detail below.

Installing a Manual Page

Mkcat is designed to make it as easy as possible to update a manual page. After the page has been reviewed for errors the installer runs:

```
mkcat page.n
```

(where *page.n* is the name of the file to format and install). This formats and installs the page, updates the "whatis" database, and builds the links required by the page.

Mkcat rejects the page if any of these errors occur:

- the formatting command fails
- no NAME section was found in the formatted text
- the NAME section was not in a known format
- the installation of the page failed

When installing a manual page, *mkcat* first analyzes the NAME section. For instance, a manual page named "names.1" that had a "NAME" header that looked like this:

```
.SH NAME
name1, name2 \— description
```

would cause *mkcat* to install the cat version as */usr/man/cat1/names.1* and build the hard or symbolic links */usr/man/cat1/name1.1* and */usr/man/cat1/name2.1*. If *mkcat* cannot parse the name section due to nonstandard formatting, it complains and quits. It has to be able to parse the NAME line correctly, because it uses that information to build hard or symbolic links when there is more than one name for the manual page.

Mkcat's original design called for a check of all the section names in the manual to be sure they are standard names. Too much variety was present in the current manual pages for this to be done.

Removing a Manual Page

Mkcat additionally provides for the removal of outdated or unwanted pages. When a product is removed from service the manual page may be removed with:

```
mkcat -D page.n
```

This will format the page, examine the formatted page for link information, remove the appropriate cat*n* links and pages, and update the "whatis" database.

This command may be given the formatted page (if, for instance the source has been lost) with the **–f** option.

Global Options to Mkcat

Mkcat has many tunable parameters that the administrator would not wish to specify for every invocation. For instance, whether to compress the cat pages is a global preference. These global preferences may be recorded at the root of the manual system in the file *.mkcat-opts*.

This file is an ASCII text file that may be viewed with a pager, or the **–V** option to *mkcat* may be used to view it with some explanatory text.

The file is created by running *mkcat* **–R**, which should be done **before** any installations are performed.

Embedded Formatting Directives

Mkcat uses the *mk*(1L) command to format the manual pages. *Mk* looks for troff-commented directives of the form "$Mkcat: ..." in the manual page and executes them. For instance, the directive shown in Figure 3 would cause *mkcat* to run the *tbl* preprocessor on the manual page and pipe this output into *nroff*. This approach allows arbitrary processing and the use of other macro packages.

```
.\" $Mkcat: ${tbl-tbl} %f | ${nroff-nroff} -man
```

Figure 3 – directive to run tbl and nroff with man macros

Maintaining consistency in the manual system.

Mkcat has a facility for checking the consistency of the formatted pages against the "whatis" database and each other. The **–L** option to *mkcat* repairs any missing links from the cat*n* directories. (Of course, there should not be any missing links if *mkcat* is used exclusively for maintenance!)

The **–Z** option to *mkcat* compresses or uncompresses files in the cat*n* directories that are out of sync with respect to the compression flag in *.mkcat-opts*.

The **–W** option to *mkcat* rebuilds the "whatis" database *in toto* from the formatted pages.

These options are used after *mkcat* is installed to give a pristine manual system for *mkcat* to work in; normally they are never used again.

Error recovery

Mkcat can rebuild the whatis database (replacing *makewhatis*) should something unforeseen happen to it. (At PUCC the most common problem with the whatis database was that someone had run *makewhatis*!)

Since *mkcat* uses *install*(1L) to install the files it modifies, it is possible to back out of incorrect installations. The **–f** option to *mkcat* allows the installer to provide an already formatted version of the page for installation (as might be found in an "OLD" directory—see install(1L) for details).

Tools *mkcat* Needs

The tools below are either available for anonymous ftp from the host "j.cc.purdue.edu" or from comp.sources.unix archive sites, or both.

install(1L) This PUCC-developed tool installs and updates system files in a controlled manner and is available in comp.sources.unix archives.

compress, zcat, uncompress (*optional*) These tools use a Lemple-Ziv compression to save a little disk space. *Mkcat* can be run without these tools, or with a different compression program if desired.

mk(1L) This program extracts shell commands from files; it has many generic in applications unto itself. See *mk*(1L)[4].

A_man, p_man, and u_man

Some System V sites have the on line manual broken down under */usr/man* into 3 subsystems. Such a division may be made using *mkcat* by supplying a **–r** option to *mkcat*. The **–r** option is usually provided by the Makefile for the product being installed. Some thought has been given to embedding "subsection" information in the manual page source (as the formatting directives are currently).

[4]This *mk* is **not** the one described in "*Mk*: a successor to *make*" in the Summer 1987 USENIX Conference Proceedings. This program predates that *mk* see "Acknowledgements".

Future work

Read "SEE ALSO" sections

It was been noted that the "SEE ALSO" section of a manual page could be mechanically parsed. *Mkcat* might be enhanced to examine the cat*n* directories for referenced pages that do not exist.

Unknown sections

The daunting variety of section names in the current manual pages may yet be overcome to allow *mkcat* to warn the installer about unknown or misspelled section names.

Further divisions

Recently it has been suggested that cat*n* directories be provided for every alphanumeric extender combination (e.g.: cat3f, cat3s, cat3x). *Mkcat* could be modified to distribute pages in this manner.

A new entry in the option file, *.mkcat-opts*, would provide a list of extenders which would be given their own cat*n* directory.

Acknowledgements

Steven McGeady wrote and posted the original *mk* program in 1983 (which has also been called *compile*). His original code has been enhanced to serve in several PUCC projects.

Jeff Smith has had significant input on the design of both *mkcat* and *install*, and provided editing assistance in the preparation of this paper.

A special thanks to all the members of Purdue University Computing Center's UNIX Group who have lived through so many revisions of this program.

Kevin Braunsdorf received his B.S. Computer Science from Purdue University in 1986. He has been working with Purdue's UNIX Group since then on a variety of system administration and user level projects. Reach him at Math Science Building; West Lafayette, IN 47907 or electronically at either ksb@cc.purdue.edu or via UUCP at purdue!ksb.

NAME

mk – detect and execute shell commands in files

SYNOPSIS

mk [**–AVachinsv**] [**–D***defn*] [**–U***undef*] [**–d***submarker*] [**–e***template*] [**–l***lines*] [**–m***marker*] [**–t***templates*] [*file*]

DESCRIPTION

Mk is a utility for detecting and executing shell commands within files. It searches through the first *lines* (default 99) of named files, looking for a *marker* (default ''Compile''). When the marker is located, the portion of the line on which it appears, after the colon and up to a NEWLINE (or an occurrence of two sequential unescaped dollar signs (''$$'')), is executed by issuing it to *sh*(1).

Normally, when the named files contain source language statements, the commands are contained in lines that appear as comments to the language processor. This is merely a convention, however, and is not a *mk* requirement.

Mk depends on the shell to do general parameter and variable expansion. However, *mk* does do some *printf*(3)–like string substitution. These substitutions produce the file name in alternate forms. The unlisted upper case letters below (DPQUX) each produce the same expansion as the lower case version, but abort the expansion if the string would be empty. They begin with a percent sign ('%'):

%f	the full name of the file specified on the command line
%r	the RCS revision file for the specified file
%d	the directory part of the specified file
%p	the prefix portion of the specified file
%q*c*	the prefix portion of the file upto *c*
%u*c*	the extension on the file after *c*
%x	the extension on the specified file, if any
%y	the file type (f,d,b,c,l,s,p) as in *find*(1)
%F	the base name of the file, no prefix
%R	the RCS revision file for the base name for the specified file
%Y*c*	the file type of this file must be *c* or abort
%%	a literal percent sign

The values of *mk*'s command line options may be substituted, using percent escapes. Each of the options, –a/A, –c, –i, –n and –v/s, may be substituted via a percent escape that begins with the option letter – e. g., ''%c'' expands to ''–c'' only if the –c option was specified in the call to *mk*. The upper case forms of these options suppress the leading dash. Other command line parameters are available as listed below.

%b	the full path with which *mk* was invoked
%e	expands to the ''–e *template*'' option, if given
%l	expands to ''–l *lines*'', if one was given
%m	the *marker* we are searching for
%o	the single letter switches supplied to *mk*
%s	the *submarker* we are searching for
%t	expands to the ''–t *templates*'' option, if given
%B	the last component of the path with which *mk* was invoked
%E	expands to the *template* option, if given
%L	expands to the *lines* option, if given

%M	the *marker* we are searching for in lower case
%O	the single letter switches supplied to *mk*, no leading dash
%S	the *submarker* we are searching for in lower case
%T	expands to the *templates* option, if given

If a percent escape fails to find the indicated data – e. g., no *submarker* was specified – it will silently cause the marked line being expanded to be rejected.

All C–like backslash ('\') escape sequences are substituted. These substitutions accommodate commands that require characters some language processors might not allow in comments.

Mk will also allow a resource limit to be set for each directive. Here are the resources that can be limited:

Resource	Description	Warning Signal
clock	wall clock seconds (done by *mk* itself)	SIGALRM
core	core dump size in bytes	no warning signal
cpu	number of CPU seconds	SIGXCPU
data	data size in bytes	no warning signal
fsize	file size of any output file in bytes	SIGXFSZ
rss	resident pages in bytes	no warning signal
stack	stack size in bytes	SIGSEGV

These resources may be specified after the complete marker and submarker. They are separated by commas from each other and the marker/submarker. Two values may follow each resource name, separated from the name by an equal sign ('=') – a warning *limit* and an absolute *maximum.* The *limit* and *maximum* must be separated by a slash ('/'). If only one value is specified, it is considered to be both the *limit* and the *maximum*.

When the directive's process reaches the *limit*, it is sent the indicated warning signal. When the process reaches the *maximum*, it is sent a SIGKILL signal. When the *clock* values are the same, *mk* waits two seconds before sending the SIGKILL.

For example,

```
C $Run,cpu=300/360,core=0: %F 1 1000
```

would send to *sh* for execution (that's the meaning of the ''Run'' marker) the base name of the file (effected with the ''%F'' substitution). The base name executable would be supplied with the arguments ''1 1000''. The CPU time limit would be 300 and the CPU time maximum, 360. The maximum core dump size would be 0 bytes.

The exit status of the *mk* command is the count of the number of directives that exited non–zero. If a command is known (or intended) to fail then the expected exit code, preceded by an equal sign ('='), may be specified after its marker (''Fail'' in the following example).

```
# $Fail=1: %F "bad args"
```

When an exit code is specified, any other exit code will be treated as a failure. The special exit code '*' can be used to force a command to always exit successfully. Any exit code except the specified one will be considered a successful exit if the specified code is prefixed with a tilde ('~'). In the following example, any nonzero exit code that ''%F'' issues will cause *mk* to issue a zero code:

```
# $NonZero=~0: %F "nonzero args"
```

The full form of an *mk* directive is

$ *marker*[(*submarker*)] [=[~]*exit–status*] [,*resource*=[*limit*][/*maximum*]] : *command* [$$]

where the *resource* may be repeated to name and set multiple resource limits.

If no *marker* line can be found in the first *lines* (default 99) of the file, a standard template file name is formed using the –t*templates* arguments (default ''/usr/local/lib/mk/_%x''). The *templates* string is subjected to *mk*'s file name substitutions (see above). *Mk* then searches the file whose name emerges from that substitution for *markers*. The special marker ''*'' matches, and is a match for, any marker or submarker.

OPTIONS

Note that options and arguments may be intermixed on the command line to change the behavior of *mk* on a per file basis – e. g., ''mk –s foo.c –v –dDEBUG bar.c''.

–A Find all the marked lines that match the specified *marker* and *submarker*, stop processing at the first command that is successful.

–D*defn* Give a definition for an environment variable. The definition must have the form *ident=value*.

–U*undef*
Remove a definition for the environment variable *undef*.

–a Find all the marked lines that match the specified *marker* and *submarker*.

–c Confirm the action before running it. This will allow the user to see the command before it is executed. A response of 'y' or 'Y' will cause the command to be run.

–d*submarker*
Look for the string ''$*marker*(*submarker*):'' in file. White space is ignored. A command line including ''–d DEBUG'' would match ''$*marker*(DEBUG):''. Submarkers in the file are ignored (do not take part in matching) if no submarker is specified. The *submarker* ''*'' matches all submarkers in the file.

–e*template*
This *template* will be searched before the file at hand. This allows *mk* to trap character special files with the %y macro.

–h Print a help message.

–i Ignore case when looking for *marker* and *submarker*.

–l lines Search *lines* lines rather than the default 99 for compilation markers.

–m marker
Look for the string ''$*marker*:'' as the delimiter for the compilation directive. The default *marker* string is ''Compile''. Note that *marker* does not include either the leading dollar–sign ('$') or the trailing colon (':'). The *marker* ''*'' matches all markers in the file.

–n Do not execute the located commands. This flag specifies that the resulting command(s) should not be executed, but only printed on the standard output.

–s Be silent. Executed commands are not output.

–t*templates*
The *templates* string produces a file name that will be searched for a *marker* line if no *marker* line is found in the file named on the *mk* call. The percent escapes described above are all available for substitution in this string, so that the template can be based on the extender of the

file being processed. The default *templates* string is ''/usr/local/lib/mk/def_%x''. More than one template option my be specified. *Mk* searches for all of them in the order they are given.

–v Be verbose. This is the opposite of –s (above), and is implied by the –**n** option (above).

–V Be extra verbose. Used only to debug *mk.*

EXAMPLES

Mk is most commonly used to produce input for the shell. The following lines might occur in a C program source file:

```
/*
 * $Compile: ${CC–cc} ${CFLAGS—O} –o %F –DFOO=1 %f
 * $Compile(DEBUG): ${CC–cc} ${CFLAGS—g} –o %F –DDEBUG %f
 * $Run: %F /tmp/test
 * $Fail=1: %F /etc/passwd
 * $Limit,cpu=100,clock=600,fsize=10000: %F /tmp/test2
 */
```

If the file were called ''foo.c'', *mk,* invoked as

```
mk foo.c
```

would send *sh* the command:

```
cc –O –o foo –DFOO=1 foo.c
```

With an invocation like

```
mk –dDEBUG foo.c
```

the command

```
cc –g –o foo –DDEBUG foo.c
```

would be sent to *sh.*

SEE ALSO

setrlimit(2), valid(1L).

AUTHORS

S. McGeady, Intel, Inc., mcg@mipon2.intel.com

Kevin Braunsdorf, Purdue University Computing Center (ksb@cc.purdue.edu)

NAME

mkcat – update and format a manual page

SYNOPSIS

mkcat [-**DIfnsv**] [-**c** *catdirs*] [-**m** *man*] [-**r** *root*] [-**w** *database*] *pages*
mkcat [-**VLRWh**] [-**c** *catdirs*] [-**m** *man*] [-**r** *root*] [-**w** *database*]

DESCRIPTION

Mkcat formats the given *pages* and installs them in the standard manual structure. *Mkcat* updates the *whatis*(1) database and creates links to the installed page to represent alternate names for the page given in the **NAME** section of the formatted page.

Mkcat depends heavily on the **mk**(1l) program to extract shell commands to format each page. The *mk* program extracts any special format directive from the first 99 lines of the manual page source file which are marked with the ''Mkcat'' marker. See the examples below.

Under the **–D** option *mkcat* deletes antiquated pages.

OPTIONS

–c*catdirs*
Specify *catdirs* as the prefix for all the cat directories. The default is ''cat'' which makes the standard manual system directories ''cat1'', ''cat2'', ''cat3'', ... etc. These are used to store manual pages from the corresponding section of the manual.

–D Delete (rather than install) the given *pages*. The associated *whatis*(1) entries are removed as well as any links.

–f The given manual pages are already formatted, simply install (delete) them.

–h Print a short help message.

–I Install the given manual page and update the whatis database. (This is the default action.)

–L Only rebuild missing links to the cat pages.

–m*man* When this option is specified *mkcat* copies the manual page source into a parallel *man* directory. This mocks the old manual system.

–n Do not really run any commands. This option outputs the approximate actions of *mkcat* in shell commands.

–r*root* Specify a user defined root for the manual system. The default is ''*/usr/man*''. This allows users to make their own manual systems in their accounts.

–s Do not output descriptions of the shell commands as they are run. This is the default.

–v Be verbose by displaying shell commands as they are run.

–V Show the current configuration of the manual system.

–w*database*
Specify a *whatis*(1) database to update. The default *database* is ''whatis'' from the *root* of the manual system. If this is a full path name *root* is not prepended to it.

–R Install a new root manual system, specify the new root with **–r**. If a options file exits in that directory it is used as the defaults for the new system.

–W Just rebuild the whatis database.

–Z Verify that all the cat pages are (un)compressed.

EXAMPLES

mkcat -v -r/usr/man -R
Install mkcat's configuration file in /usr/man.

mkcat ls.1
Update the *ls*(1) manual page and *whatis*(1) entry.

mkcat -cnew sleep.1 sleep.3
Update both *sleep*(1) manual pages and *whatis*(1) entries in the "new" cat directory.

mkcat -D catman.8
Delete the outdated *catman*(8) manual page.

mkcat -r $HOME/man myprog.1g
Update the *myprog*(1g) manual page in the user's own manual system.

.\" $Mkcat: tbl %f | nroff -man.nopage
Ultrix manual pages require the nopage macro package.

.\" $Mkcat: neqn %f | tbl | nroff -man | colcrt
This line runs the *neqn*(1) processor in addition to *tbl*(1) and *nroff*(1).

.\" $Mkcat: nroff -mlocal %f | cat -s
This manual page uses a local macro package, and doesn't use *tbl*.

FILES

/usr/man/whatis	the default *whatis*(1) database
/usr/man/cat[1-8]	the default cat directories
/usr/man/.mkcat-opts	the manual system configuration

BUGS

Under the **–n** option *mkcat* cannot always predict the links it would make.

AUTHOR

Kevin Braunsdorf
Purdue UNIX Group
ksb@cc.purdue.edu, pur-ee!ksb, purdue!ksb

SEE ALSO

apropos(1), colcrt(1), compress(1), man(1), mk(1l), neqn(1), nroff(1), tbl(1), whatis(1), catman(8)

LISA IV
October 17-19, 1990
Colo. Springs, Colorado

The Answer to All Man's Problems

Tom Christiansen – Convex Computer Corporatioin

ABSTRACT

The UNIX on-line manual system was designed many years ago to suit the needs of systems at the time, but despite the growth in complexity of typical systems and the need for more sophisticated software, few modifications have been made to it since then. This paper presents the results of a complete rewrite of the man system. The three principal goals were to effect substantial gains in functionality, extensibility, and speed. The secondary goal was to rewrite a basic UNIX utility in the perl programming language to observe how perl affected development time, execution time, and design decisions.

Extensions to the original man system include storing the whatis database in DBM format for quicker access, intelligent handling of entries with multiple names (via **.so** inclusion, links, or the NAME section), embedded tbl and eqn directives, multiple man trees, extensible section naming possibilities, user-definable section and sub-section search ordering, an indexing mechanism for long man pages, typesetting of man pages, text-previewer support for bit mapped displays, automatic validity checks on the SEE ALSO sections, support for compressed man pages to conserve disk usage, per-tree man macro definitions, and support for man pages for multiple architectures or software versions from the same host.

Introduction

The UNIX on-line manual system was designed many years ago to suit the needs of the systems at the time. Since then, despite the growth in complexity of typical systems and the need for more sophisticated software to support them, few modifications of major significance have been made to the program. This paper describes problems inherent in earlier versions of the *man* program, proposes solutions to these problems, and outlines one implementation of these solutions.

The Problem

The Monolithic Approach

One of the most serious problems with the *man* program up to and including the BSD4.2 release was that all man pages on the entire system were expected to reside under a common directory, */usr/man*. There was no notion of separate sets of man pages installed on the same machine in different subdirectories. At large installations, situations commonly arise in which this functionality is desirable. A site may wish to keep vendor-supplied man pages separate from man pages that were developed locally or acquired from some third party. An individual or group may wish to maintain their own set of man pages. Multiple versions of the same software package might be simultaneously installed on the same machine. A heterogeneous environment may want to be able to view man pages for all available architectures from any machine. Given the requirement that all man pages live in the same directory, these scenarios are difficult to impossible to support.

The *man* program distributed in the BSD4.3 release included the concept of a MANPATH, a colon-delimited list of complete man trees taken either from the user's environment or supplied on the command line. While this was a vast improvement over the previous monolithic approach, several significant problems remained. For one thing, the program still had to use the *access*(2) system call on all possible paths to find out where the man page for a particular topic existed. When the user has a MANPATH containing multiple components, the time needed for the *man* program to locate a man page is often noticeable, particularly when the target man page does not exist.

Hard-coded Section Names

Another problem with the *man* program unresolved by the BSD4.3 release was that all possible sections in which a man page could reside were hard-coded into the program. This means that while a **manp** section would be recognized, a **manq** directory would not be, and while a **man3f** directory would be recognized, a **man3x11** directory would not be.

Likewise, the possible subsections for a man page were also embedded in the source code, so a man page named something like

/usr/man/man3/XmLabel.3x11

would not be found because **3x11** was not in the hard-coded list of viable subsections. Some systems install all man pages stripped of subsection components in the file name. This situation is less than optimal because it proves useful to be able to supply both a *getc*(3f) and a *getc*(3s). Distinguishing between subsections is particularly convenient with the "intro" man pages; a vendor could supply *intro*(3) *intro*(3a), *intro*(3c), *intro*(3f), *intro*(3m), *intro*(3n), *intro*(3r), *intro*(3s), and *intro*(3x) as introductory man pages for the various libraries. However, the task of running *access*(2) on all possible subsections is slow and tedious, requiring recompilation whenever a new subsection is invented.

References in the Filesystem

The existing man system had no elegant way to handle man pages containing more than one entry. For example, *string*(3) contains references to *strcat*(3) and *strcpy*(3), amongst others. Because the *man* program looks for entries only in the file system, these extra references must be represented as files that reference the base man page. The most common practice is to have a file consisting of a single line telling *troff* to source the other man page. This file would read something like:

```
.so man3/string.3
```

Occasionally, extra references are created with a link in the file system (either a hard link or a symbolic one). Except when using hard links, this method wastes disk blocks and inodes. In any case, the directory gains more entries, slowing down accesses to files in those directories. Logic must be built into the *man* program to detect these extra references. If not, when man pages are reformatted into their cat directories, separate formatted man pages are stored on disk, wasting substantial amounts of disk space on duplicate information. On systems with numerous man pages, the directories can grow so large that all man pages for a given section cannot be listed on the command line at one time because of kernel restrictions on the total length of the arguments to *exec*(2). Because of the need to store reference information in the file system, the problem is only made worse. This often happens in section 3 after the man pages for the X library have been installed, but can occur in other sections as well.

The *makewhatis*(8) program is a Bourne shell script that generates the */usr/lib/whatis* index, and is used by *apropos*(1) and *whatis*(1) to provide one-line summaries of man pages. These programs are part of the *man* system and are often links to each other and sometimes to *man* itself. If any of the man subdirectories contain more files than the shell can successfully expand on the command line, the *makewhatis* script fails and no index is generated. When this occurs, *whatis* and *apropos* stop working. The *catman*(8) program, used to pre-format raw man pages, suffers from the same problem.

Of course, *makewhatis* wasn't working all that well, anyway. It was a wrapper around many calls to little programs that each did a small piece of the work, making it run slowly. It, too, had a hard-coded pathname for where man pages resided and which sections were permitted. *Makewhatis* didn't always extract the proper information from the man page's NAME section. When it did, this information was sometimes garbled due to embedded *troff* formatting information. But even garbled information was better than none at all. Even so, these programs left some things to be desired. *Apropos* didn't understand regular expression searches, and both it and *whatis* preferred to do their own lookups using basic, unoptimized C functions like *index*(3) rather than using a general-purpose optimized string search program like *egrep*(1).

The Solution

A Real Database

The problem in all these cases appeared to be that the filesystem was being used as a database, and that this paradigm did not hold up well to expansion. Therefore the solution was to move this information into a database for more rapid access. Using this database, *man* and *whatis* need no longer call *access*(2) to test all possible locations for the desired man page. To solve the other problems, *makewhatis*(8) would be recoded so it didn't rely on the shell for looking at directories.

Coding in Perl

When the project was first contemplated, the perl programming language by Larry Wall was rapidly gaining popularity as an alternative to C for tasks that were either too slow when written as shell scripts, or simply exceeded the shells' somewhat limited capabilities. Since perl was optimized for parsing text, had convenient *dbm*(3x) support built in to it, and the task really didn't seem complex enough to merit a full-blown treatment in C or C++, perl was selected as the language of choice. Having all code written in perl would also help support heterogeneous environments because the resulting scripts could be copied and run on any hardware or software platform supporting perl. No recompilation would be required.

Some concern existed about choosing an interpreted language when one of the issues to address was that of speed. It was decided to do the prototype in perl and, if necessary, translate this into C should performance prove unacceptable.

The first task was to recode *makewhatis*(8) to generate the new *whatis* database using *dbm*. The *directory*(3) routines were used rather than shell globbing to circumvent the problem of large directories breaking shell wildcard expansions. Perl proved to be an appropriate choice for this type of text processing (see Figure 1).

Database Format

The database entries themselves are conveniently accessed as arrays from perl. To save space and accommodate man pages with multiple references, two kinds of database entries exist: direct and indirect. Indirect entries are simply references to direct entries. For example, indirect entries for *getc*(3s), *getchar*(3s), *fgetc*(3s), and *getw*(3s) all point to the real entry, which is *getc*(3s). Indirect entries are created for multiple entries in the NAME section, for symbolic and hard links, and for **.so** references. Using the NAME section is the preferred method; the others are supported for backwards compatibility.

Assuming that the WHATIS array has been bound to the appropriate *dbm* file, storing indirect entries is trivial:

```
$WHATIS{'fgetc'} = 'getc.3s';
```

When a program encounters an indirect entry, such as for *fgetc*, it must make another lookup based on the return value of first lookup (stripped of its trailing extension) until it finds a direct entry. The trailing extension is kept so that an indirect reference to *gtty*(3c) doesn't accidentally pull out *stty*(1) when it really wanted *stty*(3c).

The format of a direct entry is more complicated, because it needs to encode the description to be used by *whatis*(1) as well as the section and subsection information. It can be distinguished from an indirect entry because it contains four fields delimited by control-A's (ASCII 001), which are themselves prohibited from being in any of the fields. The fields are as follows:

1. List of references that point to this man page; this is usually everything to the left of the hyphen in the NAME section.
2. Relative pathname of the file the man page is kept in; this is stored for the indirect entries.
3. Trailing component of the directory in which the man page can be found, such as **3** for **man3**.
4. Description of the man page for use by the *whatis* and *apropos* programs; basically everything to the right of the hyphen in the NAME section.

At first glance, the third field would seem redundant. It would appear that you could derive it from the character after the dot in the second field. However, to support arbitrary subdirectories like **man3f** or **man3x11**, you must also know the name of the directory so you don't look in **man3** instead. Additionally, a long-standing tradition exists of using the **mano** section to store old man pages from arbitrary sections. Furthermore, man pages are sometimes installed in the wrong section. To support these scenarios, restrictions regarding the format of filenames used for man pages were relaxed in *man*, *makewhatis*, and *catman*, but warnings would be issued by *makewhatis* for man pages installed in directories that don't have the same suffix as the man pages.

Multiple References to the Same Topic

A problem arises from the fact that the same topic may exist in more than one section of the manual. When a lookup is performed on a topic, you want to retrieve all possible man page locations for that topic. The *whatis* program wants to display them all to the user, while the *man* program will either show all the man pages (if the **–a** flag is

```
s/\\f([PBIR]|\(..)//g;          # kill font changes
s/\\s[+-]?\d+//g;               # kill point changes
s/\\&//g;                       # and \&
s/\\\((ru|ul)/_/g;              # xlate to '_'
s/\\\((mi|hy|em)/-/g;           # xlate to '-'
s/\\\*\(..//g  &&               # no troff strings
    print STDERR "trimmed troff string macro in NAME section of $FILE\n";
s/\\//g;                        # kill all remaining backslashes
s/^\.\\"\s*//;                  # kill comments
if (!/\s+-+\s+/) {
    #   ^ otherwise L-devices would be L
    print STDERR "$FILE: no separated dash in $_\n";
    $needcmdlist = 1;           # forgive their braindamage
    s/.*-//;
    $desc = $_;
} else {
    ($cmdlist, $desc) = ( $`, $' );
    $cmdlist =~ s/^\s+//;
}
```

Figure 1 – *makewhatis* excerpt #1

given) or sort what it has retrieved according to a particular section and subsection precedence, by default showing entries from section 1 before those from section 2, and so forth. Therefore, each lookup may actually return a list of direct and indirect lookups. This list is delimited by control-B's (ASCII 002), which are stripped from the data fields, should they somehow contain any. The code for storing a direct entry in the *whatis* database is featured in Figure 2.

Notice the check the new datum's length against the value of MAXDATUM. This is because of the inherent limitations in the implementation of the *dbm*(3x) routines. This is 1k for *dbm* and 4k for *ndbm*. This restriction will be relaxed if a *dbm*-compatible set of routines is written without these size limitations. The GNU *gdbm* routines hold promise, but they were released after the writing of these programs and haven't been investigated yet. In practice, these limits are seldom if ever reached, especially when *ndbm* is used.

Other Problems, Other Solutions

The rewrite of *makewhatis*, *catman*, and *man* to understand multiple man trees and to use a database for topic-to-pathname mapping did much to alleviate the most important problems in the existing man system, but several minor problems remained. Since this was a complete rewrite of the entire system, it seemed an appropriate time to address these as well.

Indexing Long Pages

Several of the most frequently consulted man pages on the system have grown beyond the scope of a quick reference guide, instead filling the function of a detailed user manual. Man pages of this sort include those for shells, window managers, general purpose utilities such as awk and perl, and the X11 man pages. Although these man pages are internally organized into sections and subsections that are easily visible on a hard-copy printout, the on-line man system could not recognize these internal sections. Instead, the user was forced to search through pages of output looking for the section of the man page containing the desired information.

To alleviate this time-consuming tedium, the man program was taught to parse the *nroff* source for man pages in order to build up an index of these sections and present them to the user on demand. See Figure 3 for an excerpt from the *ksh*(1) index page, displayable via the new –i switch.

```
Idx  Subsections in ksh.1    Lines
  1  NAME                        3
  2  SYNOPSIS                   22
  3  DESCRIPTION                15
  4  Definitions.               43
  5  Commands.                 338
  6  Comments.                   6
  7  Aliasing.                 107
  8  Tilde Substitution.        47
  9  Command Substitution.      28
 10  Process Substitution.      49
 11  Parameter Substitution.   645
 12  Blank Interpretation.      15
 13  File Name Generation.      87
```

Figure 3 – *ksh* index excerpt

The */usr/man/idx*/* directories serve the same function for saved indices as */usr/man/cat*/* directories do for saved formatted man pages. These are regenerated as needed according the the same criteria used to regenerate the cat pages. They can be used to index into a given man page or to list a man page's subsections. To begin at a given subsection, the user appends the desired subsection to the name of the man page on the command line, using a forward slash as a delimiter. Alternatively, the user can just supply a trailing slash on the man page name, in which case they are presented with the

```
sub store_direct {
    local($cmd, $list, $page, $section, $desc) = @_; # args
    local($datum);

    $datum = join("\001", $list, $page, $section, $desc);

    if (defined $WHATIS{$cmd}) {
        if (length($WHATIS{$cmd}) + length($datum) + 1 > $MAXDATUM) {
            print STDERR "can't store $page -- would break DBM\n";
            return;
        }
        $WHATIS{$cmd} .= "\002";  # append separator
    }
    $WHATIS{$cmd} .= $datum;  # append entry
}
```

Figure 2 – *makewhatis* excerpt #2

index listing like the one the **–i** switch provides, then prompted for the section in which they are interested. A double slash indicates an arbitrary regular expression, not a section name. This is merely a short-hand notation for first running man and then typing `/expr` from within the user's pager. See Figure 4 for example usages of the indexing features.

```
man -i ksh         # show sections
man ksh/           # show sections,
                   # prompt for which one
man ksh/tilde
man ksh/8          # equivalent to
                   # preceding line
man ksh/file
man ksh/generat # == preceding line
man ksh/13         # so is this

man ksh//hangup # start at this string
```

Figure 4 – Index Examples

This indexing scheme is implemented by searching the index stored in */usr/man/idx1/ksh.1* if it exists, or generated dynamically otherwise, for the requested subsection. A numeric subsection is easily handled. For strings, a case-insensitive pattern match is first made anchored to the front of the string, then — failing that — anywhere in the section description. This way the user doesn't need to type the full section title. The *man* program starts up the pager with a leading argument to begin at that section. Both *more*(1) and *less*(1) understand this particular notation. In the first example given above, this would be

```
less '+/^[ \t]*Tilde Substitution'\
                   /usr/man/cat1/ksh.1
```

Once again, perl proved useful for coding this algorithm concisely. The subroutine for doing this is given in Figure 5. Given an expression such as "5" or "tilde" or "file" and a pathname of the man page, *man* loads an array of subsection index titles and quickly retrieves the proper header to pass on to the pager. Perl's built-in **grep** routine for selecting from arrays those elements conforming to certain criteria made the coding easy.

Conditional Tbl and Eqn Inclusion

Several other relatively minor enhancements were made to the man system in the course of its rewrite. One of these was to include calls to *eqn*(1) and *tbl*(1) where appropriate. For instance, the X11 man pages use *tbl* directives to construct a number of tables. It was not sufficient to supply these extra filters for all man pages. Besides the slight performance degradation this would incur, a more serious problem exists: some systems have man pages that contain embedded **.TS** and **.TE** directives; however, the data between them was not *tbl* input, but rather its output. They have already been pre-processed in the unformatted versions. To do so again causes *tbl* to complain bitterly, so heuristics to check for this condition were built in to the function that determines which filters are needed.

To support tables and equations in man pages when viewed on-line, the output must be run through *col*(1) to be legible. Unfortunately, this strips the man pages of any bold font changes, which is undesirable because it is often important to distinguish between bold and italics for clarity. Therefore, before the formatted man page is fed to *col*, all text in bold (between escape sequences) is converted to character-backspace-character combinations. These combinations can be recognized by the user's pager as a character in a bold font, just as underbar-

```
sub find_index {
    local($expr, $path) = @_;  # subroutine args
    local(@matches, @ssindex);
    @ssindex = &load_index($path);

    if ($expr > 0) {              # test for numeric section
        return $ssindex[$expr];
    } else {
        if (@matches = grep (/^$expr/i, @ssindex)) {
            return $matches[0];
        } elsif (@matches = grep (/$expr/i, @ssindex)) {
            return $matches[0];
        } else {
            return '';
        }
    }
}
```

Figure 5 – Locate Subsection by Index

backspace-character is recognized as an italic (or underlined) one. Unfortunately, while *less* does recognize this convention, *more* does not. By storing the formatted versions with all escape-sequences removed, the user's pager can be invoked without a pipe to *ul* or *col* to fix the reverse line motion directives. This provides the pager with a handle on the pathname of the cat page, allowing users to back up to the start of man pages, even exceptionally long ones, without exiting the *man* program. This would not be feasible if the pager were being fed from a pipe.

Troffing and Previewing Man Pages

Now that many sites have high-quality laser printers and bit-mapped displays, it seemed desirable for *man* to understand how to direct *troff* output to these. A new option, **-t**, was added to mean that *troff* should be used instead of *nroff*. This way users can easily get pretty-printed versions of their man pages.

For workstation or X-terminal users, *man* will recognize a TROFF environment variable or command line argument to indicate an alternate program to use for typesetting. (This presumes that the program recognizes *troff* options.) This method often produces more legible output than *nroff* would, allows the user to stay in their office, and saves trees as well.

Section Ordering

The same topic can occur in more than one section of the manual, but not all users on the system want the same default section ordering that *man* uses to sort these possible pages. For instance, C programmers who want to look up the man page for *sleep*(3) or *stty*(3) find that by default, *man* gives them *sleep*(1) and *stty*(1) instead. A FORTRAN programmer may want to see *system*(3f), but instead gets *system*(3). To accommodate these needs, the *man* program will honor a MANSECT environment variable (or a –S command line switch) containing a list of section suffixes. If subsection or multi-character section ordering is desired, this string should be colon-delimited. The default ordering is "ln16823457po". A C programmer might set his MANSECT to be "231" instead to access subroutines and system calls before commands of the same name. A FORTRAN programmer might prefer "3f:2:3:1" to get at the FORTRAN versions of subroutines before the standard C versions. Sections absent from the MANSECT have a sorting priority lower than any that are present.

Compressed Man Pages

Because man pages are ASCII text files, they stand to benefit from being run through the *compress*(1) program. Compressing man pages typically yields disk space savings of around 60%. The start-up time for decompressing the man page when viewing is not enough to be bothersome. However, running *makewhatis* across compressed man pages takes significantly longer than running it over uncompressed ones, so some sites may wish to keep only the formatted pages compressed, not the unformatted ones.

Two different ways of indicating compressed man pages seem to exist today. One is where the man page itself has an attached **.Z** suffix, yielding pathnames like */usr/man/man1/who.1.Z*. The other way is to have the section directory contain the **.Z** suffix and have the files named normally, as in */usr/man/man1.Z/who.1*. Either strategy is supported to ease porting the program to other systems. All programs dealing with man pages have been updated to understand man pages stored in compressed form.

Automated Consistency Checking

After receiving a half-dozen or so bug reports regarding non-existent man pages referenced in SEE ALSO sections, it became apparent that the only way to verify that all bugs of this nature had really been expurgated would be to automate the process. The *cfman* program was verifies that man pages are mutually consistent in their SEE ALSO references. It also reports man pages whose **.TH** line claims the man page is in a different place than *cfman* found it. *Cfman* can locate man pages that are improperly referenced rather than merely missing. It can be run on an entire man tree, or on individual files as an aid to developers writing new man pages.

```
at.1: cron(8) really in cron(1)
binmail.1: xsend(1) missing
dbadd.1: dbm(3) really in dbm(3x)
ksh.1: exec(2) missing
ksh.1: signal(2) missing
ksh.1: ulimit(2) missing
ksh.1: rand(3) really in rand(3c)
ksh.1: profile(5) missing
ld.1: fc(1) really in fc(1f)
sccstorcs.1: thinks it's in ci(1)
uuencode.1c: atob(n) missing
yppasswd.1: mkpasswd(5) missing
fstream.3: thinks it's in fstream(3c++)
ftpd.8c: syslog(8) missing
nfmail.8: delivermail(8) missing
versatec.8: vpr(1) missing
```

Sample *cfman* run

The amount of output produced by *cfman* is startling. A portion of the output of a sample run is seen in Figure 6. Some of its complaints are relatively harmless, such as *dbm* being in section **3x** rather than section **3**, because the *man* program can find entries with the subsection left off. Having inconsistent **.TH** headers is also harmless, although the printed man pages will have headers that do not reflect their filenames on the disk. However, entries that refer to pages that are truly absent, like *exec*(2) or *delivermail*(8), merit closer attention.

Multiple Architecture Support

As mentioned in the discussion of the need for a MANPATH, a site may for various reasons wish to maintain several complete sets of man pages on the same machine. Of course, a user could know to specify the full pathname of the alternate tree on the command line or set up their environment appropriately, but this is inconvenient. Instead, it is preferable to specify the machine type on the command line and let the system worry about pathnames. Consider these examples:

```
man vax csh
apropos sun rpc
whatis tahoe man
```

To implement this, when presented with more than one argument, *man* (in any of its three guises) checks to see whether the first non-switch argument is a directory beneath */usr/man*. If so, it automatically adjusts its MANPATH to that subdirectory.

Not all vendors use precisely the same set of *man*(7) macros for formatting their man pages. Furthermore, it's helpful to see in the header of the man page which manual it came from. The *man* program therefore looks for a local *tmac.an* file in the root of the current man tree for alternate macro definitions. If this file exists, it will be used rather than the system defaults for passing to *nroff* or *troff* when reformatting.

Performance Analysis

The *man* program is one that is often used on the system, so users are sensitive to any significant degradation in response time. Because it is written in perl (an interpreted language) this was cause for concern. On a CONVEX C2, the C version runs faster when only one element is present in the MANPATH. However, when the MANPATH contains four elements, the C version bogs down considerably because of the large number of *access*(2) calls it must make.

The start-up time on the parsing of the script, now just over 1300 lines long, is around 0.6 seconds. This time can be reduced by dumping the parse tree that perl generates to disk and executing that instead. The expense of this action is disk space, as the current implementation requires that the whole perl interpreter be included in the new executable, not just the parse tree. This method yields performance superior to that of the C version, irrespective of the number of components in the user's MANPATH, except occasionally on the initial run. This is because the program needs to be loaded into memory the first time. If perl itself is installed "sticky" so it is memory resident, start-up time improves considerably. In any case, the total variance (on a CONVEX) is less than two seconds in the worst case (and often under one second), so it was deemed acceptable, particularly considering the additional functionality the perl version offers.

Nothing in the algorithms employed in the *man* program require that it be written in perl; it was just easier this way. It could be rewritten in C using *dbm*(3x) routines, although the development time would probably be much longer.

The *makewhatis* program was originally a conglomeration of man calls to various individual utilities such as *sed*, *expand*, *sort*, and others. The perl rewrite runs in less than half the time of the original, and does a much better job. There are two reasons for the speed increase. The first is the cost of the numerous *exec*(2) calls made via the shell script used by the old version of *makewhatis*. The second is that perl is optimized for text processing, which is most of what *makewhatis* is doing.

Total development time was only a few weeks, which was much shorter than originally anticipated. The short development cycle was chiefly attributable to the ease of text processing in perl, the many built-in routines for doing things that in C would have required extensive library development, and, last but not at all least, the omission of the compilation stage in the normal edit-compile-test cycle of development when working with non-interpreted languages.

Conclusions

The system described above has been in operation for the last six months on a large local network consisting of three dozen CONVEX machines, a token VAX, quite a few HP workstations and servers, and innumerable Sun workstations, all running different flavors of UNIX. Despite this heterogeneity, the same code runs on all systems without alterations. Few problems have been seen, and those that did arise were quickly fixed in the scripts, which could be immediately redistributed to the network. The principal project goals of improved functionality, extensibility, and execution time were adequately met, and the experience of rewriting a set of standard UNIX utilities in perl was an educational one. Man pages stand a much better chance of being internally consistent with each other. Response from the user and development community has been favorable. They have been relieved by the many bug fixes and pleasantly surprised by the new functionality. The suite of man programs will replace the old man system in the next release of CONVEX utilities.

Tom Christiansen left the University of Wisconsin with an MS-CS in 1987 where he had been a system administrator for 6 years to join CONVEX Computer Corporation in Richardson, Texas. He is a software development engineer in the Internal Tools Group there, designing software tools to streamline software development and systems administration and to improve overall system security. Reach him via U. S. Mail at CONVEX Computer Corporation; POB 833851; 3000 Waterview Parkway; Richardson, TX 75083-3851. Reach him electronically at uunet!convex!tchrist or tchrist@convex.com.

LISA IV
October 17-19, 1990
Colo. Springs, Colorado

Life Without Root

Steve Simmons – Industrial Technology Institute

ABSTRACT

"Power corrupts."
– I. Amin

Often the people most qualified to perform certain system administration tasks are not necessarily qualified to have root access in general. This paper will discuss the rationale and methods for having non-root accounts do some types of systems administration. We will discuss two subsystems which we are currently administering without root and apply that experience to suggest some general rules.

Rationale

"With great power comes great responsibility."
– R. Nixon

A significant number of system break-ins are caused by incautious use of the root account. Other times an operator or new administrator inadvertently destroys sensitive system setups while using the root account for some completely unrelated purpose. Both of these have their root cause (pun intended) in UNIX's inability to grant privileges greater than ordinary user accounts without granting total access.

While there is no general solution to the problem, it can be mitigated by reducing the number of subsystems which require root access. This requires either removing the subsystem (not usually acceptable) or modifying it such that a normal user account can administer it. We have had good luck with the latter, successfully administering news and uucp without using root except for creating accounts.

Along the way we've fallen into a few holes and established a few rules for keeping things running properly. We also have some suggestions for present systems designers.

Using non-root accounts to administer such subsystems as news may have some additional benefits. In many situations management may be unwilling to have any fraction of administrative time devoted to such "frivolous" pursuits. Or they may allow it, but put it at such a low priority that it effectively is never done. Using a non-root account enables one to hand off responsibility to some highly motivated third party who will do the administration on his or her own time without giving that third party special privileges.

Practical Experience

"What have you done for me lately?"
– S. Goodman (deceased)

To date we have fully implemented two subsystems on which we can do rootless administration. The first of these, the network news, is described in some detail. The second, uucp, was significantly easier and is only briefly described.

News

"The News Is What You Need."
– Detroit News Adv.

Overview

We have successfully administered both Bnews 2.11.14 [1] and Cnews [2] without using root. This turned out to be surprisingly easy; in retrospect we were fortunate in choosing news as the first subsystem to do this with.

Working as root, a standard installation of Bnews was done. (A later installation of Cnews was done, requiring work under 3 different accounts.) A new user account, news, was installed on the system. The home directory for this system was installed as **/usr/lib/news**.

It was not judged sufficient that the news account simply be able to maintain the system; we wanted it to be able to apply and install patches, debug systems, etc. Thus the first step was to find all the files needed by news and make sure they were writable by news.

We did not desire that news be able to install files in such sensitive directories as **/bin**, etc. This would have opened up the entire system to any errors or malice by the netnews administrator. Instead we decided that initial installation must be done by root, but all files installed would be owned by news.

News was installed (actually re-installed, since we were already running news) and we began. Immediately we ran into problems.

Shell scripts run by *cron*[3] tend to create files owned by root. The news user was not able to modify these files, making administration difficult. We considered a number of approaches to deal with this, and actually implemented two of them. Both worked, but the second was definitely the better method.

We considered and rejected modifying all the news scripts to explicitly set file ownerships, groups, and permissions. This would permit *cron* to run as root while not causing problems with news administering news. The modified scripts would be owned by news, letting the admin modify the scripts without having to modify **crontab**. While this probably would have worked, we rejected it as making us diverge too far from standard news setups. The fixes and scripts made available by the general community of news managers are invaluable in running news. Wholesale modification would have made tracking those fixes tedious and error-prone.

Our next attempt focused on using *cron*. We modified all the **crontab** entries so that they did a *su* to news before actually invoking the maintenance script. Note that Berkeley *cron* has a feature to do this automatically, but we were not running BSD everywhere and desired to keep **crontabs** as similar as possible. While this worked, it resulted in a rather baroque **crontab** (assuming that's not redundant) and still left the problem that any modification to **crontab** itself must be done by root.

At this point we got lucky in the person of Paul Vixie. Paul was looking for someone to beta test a new PD implementation of *cron*, *cron2*[4]. *Cron2* works rather like System V *cron*[5] but has additional small but powerful features which have eased its use. Like System V *cron*, it is significantly better about error handling, security and reporting than other *crons*. Most important, it allows users to have individual **crontabs** which are always executed as the user, not as root. Vixie *cron* worked fine even in beta release (it has been submitted to comp.sources.unix as of this writing) and quickly went into general release on our systems, displacing the vendor-supplied crons.

Two particular features of Vixie *cron* must be mentioned. First, it allows definition of a user to be contacted whenever a script has unexpected (untrapped) output. Thus the *usenet* alias can continue to receive all incoming messages (presumably both the prime system admin and the news admin would be interested in these), while ordinary *cron* messages would go strictly to the news admin.

Second, it allows an alternative PATH to be specified for all invocations from a **crontab**. This permits one to place all administrative scripts in a separate directory not in the standard user path but does not require 'hard paths' in the shell scripts. This reduces system clutter and makes the scripts more readable and maintainable. It also increases system security in several ways. First, an explicit path defeats several known security holes; second, one can make the script directory inaccessible to ordinary users.

This done, we proceeded to use the system. To our pleasant surprise it worked almost immediately, with most of the tuning done on the shell scripts.

When Bnews patches 15-17 came out, we were able to install them over the existing software without requiring root. Similar results were obtained with *rn*[6] and *nn*[7]. When we converted to NNTP-based news, root was required only to create new programs in areas such as **/bin**.

When the time came to convert to Cnews, we were able to administer it as news immediately. No script or program modifications were required. The installation method for Cnews is rather baroque, and will require root intervention for the foreseeable future.

Our news administration has been done on BSD 4.3 and System V.2 without root for approximately a year; on Ultrix for 5 months. It is now being ported to SunOS 4.1.

UUCP

"Uucp is a dead protocol."
– P. Honeyman

With news in place, the next task was uucp. This too proved easier than expected. Building on our lessons from Vixie *cron*, it took only odd moments across a week to convert to non-root administration.

One might ask why bother with uucp? We have between 45 and 60 uucp connections at any given time, and are regularly adding or deleting them. Often queues or locks need to be reset, and we run an extensive set of trouble-watching scripts.

Uucp administration also maintains and updates our *pathalias* [8] database and maps. This required some careful division of responsibility between news and uucp, but has been running as a stable system for some time.

Creation and deletion of uucp login accounts still requires root intervention, but all other work is now done by the uucp account. This system has been in production use on BSD 4.3 and System V.2 for approximately a year. As of this writing we are converting it to HDB uucp on SunOS 4.1 and Ultrix 4.0.

Other Possibilities

"Anything's possible."
– U. Geller

We are convinced by our initial success than many subsystems could be broken away from root. Other more pressing tasks have prevented us from addressing them, but we hope to return to them by the end of this year. Some of these are listed below, with preliminary thoughts on difficulty.

Line Printers

Like news and uucp, it should be possible to administer the line printer spooling system without requiring root. Informal contacts suggest that *plp*, the public domain replacement for BSD-style spooling, could easily be used for this purpose.

There are additional difficulties which come from the overuse of the daemon account, see below for more on this.

Mail

We are convinced it is possible to administer mail without requiring root. What we are not yet convinced of is whether it is worth the effort. Such things as alias files, sendmail or smail configurations, etc, could be managed from an unprivileged account with relatively little effort. Going beyond that may require massive effort, and the early returns say it would probably not be worth it for a single site.

Mail also suffers from the overuse of daemon.

Prescriptions For Rootless Subsystems

"Just follow these 327 simple steps..."
– Too common

To be a candidate for non-root administration, a subsystem must be relatively self-contained and require few or no suid root programs. Once those guidelines are met, a relatively small number of rules will help get the job done.

Distribute Responsibilities Carefully

Dividing work between the news and uucp accounts proved interesting. Consider the case of comp.mail.maps and *pathalias*.

Once uucp maps have been unpacked for *pathalias* and queued to remote systems there's no point in keeping them in the news system. But who should be responsible for what? Currently the news system has responsibility for unpacking the maps and creating the pathalias database. This is because any other solution would have required giving uucp permission to delete or modify files owned by news. But the pathalias database in properly the province of uucp, not news. Since we only maintain one map set for our entire network, we build several databases on one central system and distribute them as needed.

Our initial selection is functional, but probably not optimal. Doing this as news requires a news account on all systems, regardless of their having news. A better breakdown would be to have news unpack and install the maps, then perform expires on comp.mail.maps after enough time has passed to feed our neighbors. Uucp should be creating and updating the databases.

One Account, One Purpose

As one possible solution to above, we considered having a single account to manage both systems. We rejected this on two grounds. First, it violated the rule of keeping separate systems separate. Second, it could lead to errors when suddenly the entire news system seems to be owned by uucp or vice-versa.

There's No Place Like Home

Location of homes for uucp and news accounts turned out to be important. Putting them in **/usr/lib/news** or **/usr/lib/uucp** quickly turned out to be a bad idea. Any account has a lot of files associated with it, and having those files appear in the system management areas quickly leads to confusion. The best solution was a subdirectory under each of these that was the home for the account.

We were concerned that having uucp someplace other than **/usr/spool/uucp** or **/usr/spool/uucppublic** might lead to problems with files being delivered to unexpected places; in practice this has not been a problem.

Get A Real Cron

You can run without root using even the oldest, creakiest *cron*. It's just a lot easier with a good one. Standard system V *cron* is the minimum acceptable for our systems, Vixie *cron* is the preferred solution.

Never Use Root!

Once you've converted to non-root usage, never get lazy and do something as root. More than once this has broken things in strange and subtle ways. The most common problem comes from running some news or uucp script as root. The script leaves behind files owned by root rather than the administrative id. These files are usually no longer usable by the subsystem, causing lots of interesting problems. In general, once you've converted a subsystem to non-root administration, don't go back.

Daemon Considered Evil

One of the difficulties we see in converting some systems to rootless administration is the gross overuse of the daemon account. Some mail logs use it; the line printer spooler uses it; everybody and their worm's mother uses it. This might have made sense once, but no longer does.

Giving a daemon user account would award the owner power over a large number of different subsystems. This is not acceptable for reasons of security, safety, and compartmentalization. Any subsystems we administer from non-root accounts will not use daemon.

Future Directions

"This is another fine mess you've gotten me into."
– R. Reagan

It is not difficult to design subsystems such that rootless administration is possible. Such systems as backups, batch processing, network management, etc, could very easily be administered without root. Given such tools as suid root *perl* scripts, even tasks like user account management could be done without requiring root.

If this is true, why doesn't it happen more often? The primary reasons seem to be:

- lack of forethought on the part of the system designers;
- lack of documentation on how to do so;
- lack of time or willingness on the administrators part.

All of these are solvable problems; we hope this paper has at least partially lit an unnecessarily dark corner.

References

"Read the f-f-f-f-fine manual."
– All of us

[1] M. Glickman, M. Horton, R. Adams, *"USENET Version B Installation"* from *"UNIX System Managers Manual, 4.3 Berkeley Software Distribution"*, Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, April, 1986. SMM:10.

[2] G. Collyer, *"Installing "C News" Network News Software,"* CNews documentation. Posted to comp.sources.unix volume 19, 1989.

[3] Cron(8) manual page, from *"UNIX System Managers Manual, 4.3 Berkeley Software Distribution"*, Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, April, 1986.

[4] P. Vixie, unpublished cron2 documentation. With cron2 beta release software, December 1988.

[5] Cron(8) manual page, *"SunOS Reference Manual, Volume III"*, Sun Microsystems, Revision A, March 27, 1990.

[6] L. Wall, rn installation documentation and scripts. Posted to comp.sources.unix, volume 1, 1985.

[7] K. F. Storm, NN usage and installation documentation. Posted to comp.sources.unix volume 22, 1990.

[8] P. Honeyman and S.M. Bellovin, *"PATHALIAS or The Care and Feeding of Relative Addresses,"* "USENIX Conference Proceedings," Atlanta, Summer 1986.

Steve Simmons is a graduate of the University of Michigan, and has done UNIX-based development at Bell Northern Research, Schlumberger Technologies, and ADP Network Services. He is currently the UNIX systems manager at the Industrial Technology Institute and a consultant. His publications include music, humor, essays, and software. He has published no intentional fiction. Reach him at Industrial Technology Institute; P. O. Box 1485; Ann Arbor, MI. 48106 or electronically at scs@iti.org.

LISA IV
October 17-19, 1990
Colo. Springs, Colorado

Tinsley Galyean – Brown University
Trent Hein and Evi Nemeth – University of Colorado

Trouble-MH: A Work-Queue Management Package for a >3 Ring Circus

ABSTRACT

Efficient management of users' problem reports and requests is a fundamental system administration task. This paper presents a work-queue management solution (Trouble-MH) based on the Rand MH Message Handling System. Trouble-MH allows a system administration team to easily track and resolve users' "trouble" reports.

The Problem

Efficient communication between a site's users and its system administration group is the key to a happy campus. At a small site, users know their system administrator (SA) on a first name (or at least login name) basis, and thus reporting a system problem is as simple as chatting next to the coffee machine or sending a mail message to the SA.

At larger sites, however, life gets much tougher. In our case, we needed a way for the system administration staff to manage the queue of incoming system "trouble" reports as well as requests for general information, system improvements, etc. We call the queue which collects all of these messages the "trouble queue."

When we set out to design a package to manage this queue, we had these things in mind:

- It is impossible for the user community to keep track of the names of members of the system administration group. Our Engineering Research Computing Center is staffed by 10-20 SAs, many of whom are students, and thus come and go with the school year.
- In order to simplify status tracking of a task as well as provide users with a personal reply, mail acknowledging the problem should be from a specific individual rather than a program or generic user like "sysadmin" or "trouble."
- Because our system administration group is large and spread out across 750,000 sq ft, it must be easy for any system administrator to electronically determine the current status of a task, including who is working on it already and what progress they have made.
- Multiple SAs need to be able to view and resolve items in the queue simultaneously, without concerns like the need for a lock on the entire queue which using /bin/mail might introduce.
- The solution for trouble queue management needs to be easy-to-learn, easy-to-use, and easily integrated with an SA's personal mail handling methods.
- It is always easier to provide extensions to an existing mail management system rather than to write a new one from scratch. While the finished product is not always perfect in this case, if the underlying system is powerful enough, it can grow as needs change.
- SAs need to be able to resolve items in the trouble queue from their own workstation or terminal at home without being forced to login to some overloaded central queue-management host.
- Each SA must login as himself and be accountable for his actions; therefore no generic "sysadmin" login should be used.

The Base Solution

We decided upon a solution based on the Rand MH Message Handling System. This system, hereafter referred to as MH, is publicly available via anonymous ftp from a number of sites. This paper assumes the reader is familiar with standard MH commands. Readers who are unfamiliar with MH are referred to the MH documentation [3].

MH is well-suited for managing a multi-access "trouble queue" because:

- The trouble queue can be managed as an MH "folder." SAs who normally use MH for their personal mail box can easily switch between their personal mail folder and trouble mail (which appears as a normal mail folder).
- The MH trouble folder can be kept on one host but exported via NFS to other hosts. (Please note, however, that MH performs poorly on memory-starved diskless workstations.)
- Since MH operates on a one-message-per-file

basis, there is no need to lock the entire queue for common operations.

- Trouble mail readers can be added without requiring root or other special privilege access. (access to the trouble folder, if necessary, can be restricted by creating a "trouble" group.)
- SA's can resolve items in the queue from their own account, adding that "personal touch" to the reply the user receives.
- The MH system is well-documented.

The Trouble-MH Gameplan

First of all, let's consider how we'd really like to see things happen:

1. Desperate user mails "trouble" describing a problem using their favorite mail sending technique and editor (no need to learn how to use some special trouble reporting command).
2. User's message appears in SA trouble queue.
3. On-duty SA examines message. If the issue can't be resolved immediately, an acknowledgement is sent back to the user saying that the system administration group has received the message and is looking into it. This acknowledgement is recorded in the message so that other on-duty SAs don't also acknowledge it, thereby drowning the user in content-free mail.
4. As the status of the problem changes, the user is notified and that status report is also recorded in the message. This allows any of the SAs or their managers to see the status of the item and its resolution to date.
5. When the issue finally gets resolved, the user is notified and the item is removed from the queue.

Implementation

After much thought, we decided that it was not sufficient for our purposes to use the MH system strictly "as is." We also did not have the resources to do the project right; so a combination of shell/C hacks were done as a prototype. As with many quick hacks, a useful tool emerged. Trouble-MH now in use by a handful of system administration groups at the University of Colorado to manage a variety of work queues.

For convenience, a user "trouble" was created on one of the machines belonging to our system administration group, and an alias was set up in the aliases file (either /etc/aliases or /usr/lib/aliases) which directed all mail sent to "trouble" to this account. This account has a .forward file which directs incoming mail through the *slocal* program and into the trouble folder.

In each SA's MH directory (usually ~/Mail) there is a symbolic link which points to this common trouble folder either on the local machine or an NFS mounted partition. To list the messages in the trouble folder, an SA can type the standard MH command:

```
    scan +trouble
```

We want the status/acknowledgement information to be stored in the body of the message in the queue. In order to provide this functionality, a new command, *stat* was introduced. *stat* invokes *repl* with these lines appended to the end of the message:

```
----------
Status: by <your-login> <date>
----------
```

After you are finished editing, *stat* appends the text you typed below the Status: line to the end of the message, leaves it in the queue (for others to see later), and also sends it to the user reporting the problem. A typical message in the trouble folder looks like:

```
(Message trouble:10)
To:      trouble@boulder
From:    ajsh@wild.colorado.edu
Subject: Bug in Sun Fortran 1.3
Date:    Wed, 18 Jul 90 18:06:01 MDT

Is this a bug in Sun Fortran 1.3
compiler?

Here's my floating point option, my
fortran program tst.f, how I compile
it, and the results.

   <code example>

What do you make of it? -- Andrew

----------
Status: by huntert Thu Jul 19 10:50:13 1990
----------
Trouble has received your message, we'll
look into it asap.

----------
Status: by hardt Thu Jul 19 16:40:27 1990
----------
I have looked at this with him and we
have gotten it down to a simpler example.

----------
Status: by kiefer Mon Jul 23 11:44:56 1990
----------
I have verified that this is in fact a
bug in the compiler and have reported it
under our software support contract.  I'm
awaiting a reply from Sun ...
```

Another command, *res*, which stands for "resolve message", was also created. *res* uses *repl* to reply to the message and then uses *rmm* to remove it from the queue. One could also refile the message in an old-trouble folder for later reference instead of just removing it.

Thus, *stat* and *res* are front-ends to repl and provide a uniform reporting and tracking mechanism.

After about a year of use, we noticed that SAs spent a lot of time sending the same reply to multiple messages about the same topic (a major print server was down, so ALL the secretaries sent mail about it, for example.) In order to deal with situation, a third command, *glom*, was added. *glom* takes a list of messages and combines them into a single message, with copies going to all the appropriate people, so that a single reply suffices.

Other Advantages

One of the side effects of using Trouble-MH is that each message can serve as a teaching tool for new SAs. It not only shows how to solve a particular problem, but also is an example of the tone of response you expect the SA staff to use in answering users complaints. As such it allows managers to identify incorrect answers or unprofessional responses and bring them to the attention of the errant SA.

Queue-MH

The package was originally called Trouble-MH and was intended for SA trouble mail use only. After a bit of use, it became clear that Trouble-MH was useful in other arenas. In our own group, we soon had Operator-MH (for the operator backup/restore queue) and Wiring-MH (for the wirers). We merged them into one generalized package called (for lack of a better name) Queue-MH.

Acknowledgements

Trouble-MH was originally designed and implemented by Tinsley Galyean and Trent Hein (under the supervision of Bob Coggeshall) in October, 1988. Functionality improvements ("glom", in particular) have been added by John Hardt. Barb Dyker added queue generalization and management documentation. Most recently, Andy Kuo cleaned up the scripts and improved locking.

Source Availability

You can obtain the source code and documentation for the package described in this paper via anonymous ftp from boulder.Colorado.EDU [128.138.240.1] (pub/queuemh.tar.Z). Although we don't officially support this distribution, and are actually somewhat embarrassed about it, we use it daily. Please report any bug fixes or suggestions you have to "systems@boulder.Colorado.EDU."

References

[1] Dyker and Hein, "Using Queue MH," May, 1990.

[2] Dyker, "Managing Queue MH," May 1990.

[3] The Rand MH User's Manual.

Tinsley Galyean received a BS in Computer Science from the University of Colorado, and recently received a Masters degree from Brown University. This fall, Tinsley is continuing his education at MIT. Contact him electronically at tag@media-lab.media.mit.edu .

Trent Hein is both on the staff and a student at the University of Colorado, Boulder. Trent spent this past summer at UC-Berkeley working on the 4.4 BSD port to the MIPS architecture and various other things. Contact Trent at University of Colorado, Boulder; Department of Computer Science; P. O. Box 430; Boulder, CO 80309-0430 or electronically at trent@boulder.Colorado.EDU .

Evi Nemeth spent the past year at Dartmouth College on leave from the University of Colorado, Boulder, where she is on the faculty of the Computer Science department. She has recently co-authored a book on system administration: *The UNIX System Administration Handbook*, published by Prentice Hall. Contact Evi at University of Colorado, Boulder; Department of Computer Science; P. O. Box 430; Boulder, CO 80309-0430 or at evi@boulder.Colorado.EDU electronically.

LISA IV
October 17-19, 1990
Colo. Springs, Colorado

A Console Server

Thomas A. Fine and Steven M. Romig –
The Ohio State University

ABSTRACT

Most computers nowadays include some sort of "console terminal" from which one can perform certain special administrative tasks, such as booting the machine, or running diagnostics. Some programs also write error messages to the console terminal. Unfortunately, these consoles take up valuable space, often in rooms with special environments where space is at a premium. Many people use CRTs as console terminals, which usually means that they can not make a record of the messages that have appeared before the latest 24 lines. Finally, the location of the console terminals tends to be inconvenient.

We had 30 console terminals piled on desks in our machine room, terminals which took up a considerable amount of space. We replaced most of these console terminals with a spare Sun 3/180 outfitted with 32 serial ports and a custom software package called "the console server". The console server saves us space in our machine room, logs output from all of the console ports, and provides convenient access to the console ports on the machines that it serves.

In this paper we will describe the reasons for undertaking this project, and the design and implementation of the software.

Introduction

Our environment at Ohio State has been growing rapidly over the past three years. We found ourselves faced with several problems relating to the console terminals in our machine room. The first problem we faced was that we were running out of space in the machine room. Most of our servers are Suns, and there is no convenient place to put the terminals on the Sun cabinets (we were using h19 terminals). We had 30 of these terminals piled four high in the machine room, which took up about 80 square feet, with no room left for any new machines. This arrangement also made it difficult to use most of the consoles because they were either on the floor, or at shoulder height.

Another problem we wanted to solve was the lack of logging with the h19 terminals. Often a machine would crash, or some other problem would occur, and by the time someone got into the machine room, any relevant messages were already scrolled off of the screen. We did not want to use printing terminals, since they would have made our space problem much worse (they can not be stacked), and they are prone to periodic mechanical failures. Of course, syslog does provide some logging, but it is by no means complete.

A third problem was convenience. As mentioned above, the arrangement in the machine room made it difficult to use many of the terminals. Their location was also very inconvenient, because of the need to run to the machine room every time a problem arose. We could not just move the consoles because many times one needs to be near the machine while at the console. We could have added consoles with Y connectors, but this would have been a cabling nightmare, and would not have helped either our space or logging problems.

Our solution was to replace all the terminals with a single machine servicing all the console terminal lines, and software which provides logging and direct, secure network access. The machine, the console server, is a Sun 3/180 with 32 additional serial ports (two alm2 cards). The software consists of a server program which does the logging and handles connections, and a client program used to connect to any desired terminal.

This solves all of our problems nicely. We immediately freed up about 60 square feet in our machine room (enough space for at least six more servers). The logging provided is complete, as long as the server is up. The improvement in convenience is enormous, providing access to our file server consoles and log files from anywhere in our network, as well as from home (with a modem).

Environment

Our environment is primarily a network of Sun workstations, served by 23 Sun file servers which are hooked to our console server. We also have four Pyramids, an HP file server, an Encore Multimax, and a BBN Butterfly attached to the console server. The console server itself does not depend on any other machines, and only handles administrative tasks.

Design

The software consists of a server program and a client program. The server program logs all the activity from the consoles, and handles network connections from the client program. The client program is used to request a connection to a console from anywhere on the network.

The client program requests a given console line by the name of the file server, and can request one of four commands on that console line:

1. Attach (two-way connection)
2. Spy (read-only connection)
3. Force (two-way connection, superseding an existing two-way connection)
4. Who (returns who is connected to the requested console line)

In addition to the ''who'' command above, one can request a list of all connections everywhere, by requesting connection to a server named ''who''. This last was added as an afterthought, and does not use the normal connection protocol used for other connections.

Escape sequences are provided for controlling the state of the connection. This includes switching between read-only and two-way connections, toggling flow control, and disconnecting. An escape sequence is also provided to halt Sun file servers. All of the escape sequences are handled by the server program.

A simple configuration file is read when the console server is started. This file contains file server names, their log file, the serial port they are attached to, and a group number which determines which servers will be handled by each server process (described below).

Server Program

The server program forks off several children, with each child process handling a small subset of the consoles. This was done primarily because a single process would run out of open files quickly (each terminal uses two files, and each connection to a terminal uses one). Alternatively, with a separate process for each console, we were worried that the excessive number of processes would bog down the machine. In our current configuration, we use eight child processes, each handling four or five console lines.

The parent process remains to serve as a port lookup for the other processes. Each of the children listens on a separate port for connections, but only the port used by the parent process is known to the client program. The client must first connect to the parent process and request the port number for the desired server, then connect to the child process at that port.

The server program handles all connection states, keeping track of which connection is two-way for each of the console lines. It is also responsible for breaking connections when requested with an escape sequence from the client program (typed by the user).

Client Program

The client program is a very straight-forward piece of software. It connects to the server software as described below and, if it succeeds in connecting, it simply passes all data from the keyboard to the console server, and all data from the console server to the screen, until the console server breaks the connection.

Connection Protocol

The following is a simple version of the connection protocol:

1. The client program connects to the parent process of the server program, and sends the name of the server requested by the user.
2. The server program responds with either a port number (sent in ascii) or the message ''Server not found'', and breaks the connection.
3. If the client program receives the error message, it displays the message, and exits. Otherwise, it requests a password from the user, connects to the new port number, and transmits the user's name, the password, the source machine, the command (two-way, read only, forced, who), and the file server name.
4. The server program (child process) checks the password, and if valid, it completes the connection, sending ''ok'' to the client program. If it is not valid, the server returns the message ''Sorry'' and breaks the connection.

   Note that if the command is who, password checking is skipped.

Security

The console terminals can do special things on servers (like halting, booting in single user mode) so we were somewhat concerned about security. The password required is a yellow pages password, so the level of security provided by this method is as good (or as bad) as the rest of the system. We would eventually like to use Kerberos to authenticate users in a more secure fashion.

Implementation Issues

The initial effort required for the software was minimal. The first version (with few features) was written in just a few days. Fine tuning has been a much longer process which is not yet (and may never be) complete. The following covers problems we encountered and features added (in no particular order).

- We immediately added flow control, because we found that there was significant data loss when the output was going to a slow terminal (like the Sun frame buffer terminal emulator). Faster terminals (a real CRT, or an X windows terminal emulator) rarely had a problem, but the flow control was still necessary. An escape sequence was also added to toggle flow control on and off, primarily so emacs could be used without rebinding control-S.

- The majority of our file servers are Sun 3s. These machines do not have an ascii sequence used to halt the machine. Instead (if the console is on a serial port) they use a terminal line break. We had to add an escape sequence that would tell the console server to generate a line break. That was simple enough, but this also presented another problem: we wanted to be able to remove the cables from the console server at various times. Unfortunately, the suns also interpret a drop in the Data Terminal Ready line as a request to halt the server. We had to modify the file server end of the cable to maintain DTR even when the cable was unplugged on the console server end.
- Four of our file servers are Pyramids. Unfortunately, these machines are convinced that their terminals are Wyse terminals (it is built into their microcode). There was no good way around this, so we have Wyse terminals scattered around in convenient places. We also wrote a program that will filter out the worst of the Wyse sequences, so that if a Wyse terminal is not available, you can scrape by with whatever you have.
- We realized that it would be convenient to see the last few lines of output from a console's logfile after connecting. We added an option to display the last twenty lines from the log after connecting. This is sometimes not enough, and we may add escape sequences to replay arbitrary amounts of the log file.
- It is possible to create a loop of two or more console connections, which will continually circulate the connection message through the loop (and into all involved log files). If a disconnect sequence is then sent by the user, it only reaches the first connection, but leaves the loop intact. There is currently nothing to solve this problem, but it has only occurred (accidentally) once.
- The log files can sometimes become rather unwieldy. We have them on a file system with lots of space, so that has not been a problem, but it is hard to get useful information out of a 24 Meg log file. Ordinarily the log files grow slowly, but when there is a problem (like a root file system becomes full), the file server can generate an amazing volume of messages in a short period of time. We haven't gotten around to providing automatic truncation of these files, but it should not be a big problem.
- We needed the log files to contain time stamping. We used a simple shell script that wakes up every hour and prints the date in each logfile. This provides time stamping without interrupting activity on the console.
- We were initially worried about the speed of the console server during excessive use (many connections, or many machines rebooting) but we have not seen a problem like this so far.
- If the console server ever crashes badly, and we have a hard time getting it back up, we will have to deal with all the other file servers by plugging cables into a spare terminal. Since a bad crash is likely to come from a power outage, this could be a serious catastrophe, as most of the servers would probably need immediate attention. It has not happened yet (knock on wood).

Conclusion

The console server really works well. It has done everything we hoped it would, and has not had any serious problems.

The improved convenience has been the biggest payoff from the project. The ability to fix a problem from home (or from Anaheim, California) has saved a great deal of time, gasoline, and sleep. And in the office it enables staff members to become even more desk-bound.

There have been a couple of other benefits that have not been mentioned above. Since the console server software does not heavily load the machine itself, it is an ideal machine to handle other administrative tasks (yellow pages master, backups, etc.). With all of these uses, the console server is a very cheap alternative to CRT terminals.

Tom Fine is a student programmer for the CIS Department at The Ohio State University, where he is working towards a B.S. in computer science. Reach him at Computer and Information Science Department; The Ohio State University; 2036 Neil Avenue Mall; Columbus, Ohio, 43210 or electronically at fine@cis.ohio-state.edu .

Steve Romig is the software staff manager for the CIS Department at The Ohio State University. He received a B.S. degree in applied math from Carnegie Mellon University in 1982 and is slowly working toward an M.S. degree in computer science at Ohio State. His main professional interests are in simplifying and automating system administration tasks and in computer security. Reach him by U. S. Mail at Dept. of Computer and Information Science; 2036 Neil Avenue Mall; Columbus, OH 43210. Reach him electronically at romig@cis.ohio-state.edu .

LISA IV
October 17-19, 1990
Colo. Springs, Colorado

Network Monitoring by Scripts

Katy Kislitzin – Computer Sciences Corporation

ABSTRACT

This paper describes several network monitoring tools written in *sh* and *gawk*. These tools include a *ping*-based diagnostic report, a network availability metric and an *ftp* throughput report.

The NAS Environment

The goal of the Numerical Aerodynamic Simulation (NAS) Facility at the NASA Ames Research Center, is to be a pathfinder in numerical aerodynamic simulation methods. One aspect of achieving that goal is a commitment to having the fastest machines in the world and a network which can support them.

Currently the NAS high speed processors consist of a Cray Y-MP and a Cray-2. In addition, there are several highly parallel machines including a Connection Machine and an Intel Touchstone Gamma Prototype. The support machines include several VAXen and an Amdahl mass storage system.

Locally, most scientific users have high end Silicon Graphics workstations, and the support staff has a combination of Silicon Graphics and Sun workstations. All NAS workstations are operated in a diskful configuration with one or more local disks. In addition, Sun fileservers provide system and local programs to all workstations. In all, there are over 230 workstations and fileservers, a number of Macintoshes running Appletalk connected to the NAS local area network (LAN) via Kinetics boxes, and a small number of PCs and other machines. All NAS machines use some variant of Unix, with the exception of the Macintoshes and a few unsupported personal computers.

As the NAS is a national resource, most users are not local. Many sites access the NAS using the Internet. Ames Research Center has direct connections to BARRNet, NSInet, and the MILNET, among others. NAS is one hop away from those connections. In order to provide better service to locations which have large research projects, the NAS has a bridged WAN consisting of 25 56 Kbps lines, one 896 Kbps line and 2 T1 (1.544 Mbps) lines. Some of these sites have routers which connect to their LAN at the remote end. However, in most cases the connections are made using secure bridges which will only allow traffic between specific hosts at the remote site and specific NAS hosts.

In addition to the long haul dedicated lines, network media includes ethernet and twisted pair ethernet (10 Megabit/sec), Hyperchannel (50 Megabit/sec per trunk), and Ultranet (1 Gigabit/sec). Up until January, the NAS LAN consisted of a bridged class B network for ethernet and long haul links, and separate class C networks for Hyperchannel and other media networks. This past winter, the NAS LAN was converted to a single class B network with an 8 bit subnet mask and 5 main subnets. Five more subnets will be split off in the near future, primarily to isolate traffic between workstations and fileservers.

To take advantage of higher speed media, a major reconfiguration of the NAS LAN is in the design and testing phases. Possibilities include a medium speed research network in the Gigabit/sec range, primarily for large data transactions with the high speed processors. The research network will be supported by an approximately 100 Mbit/sec backbone, which will feed ethernet segments for the fileservers and workstations. A concurrent project is underway to upgrade the long haul network. The current bridges will be replaced with routers and all lines will be upgraded to T1 initially. Eventually the plan is to upgrade to T3 speeds (45 Mbps).

Monitoring Goals

As the network grows more complex, it becomes crucial to have tools which help to understand and manage it. Hence, several tools were developed to meet the following goals:

1. Notify staff that part of the network is isolated or that the quality of service is degrading towards an unacceptable level.
2. Flag router failures.
3. Provide a network availability metric.
4. Simulate throughput experienced by users between their workstations and the high speed processors (the Crays).

To accomplish the first three, a *ping* based monitoring and reporting system, collectively referred to as **pingit**, was developed. To accomplish the fourth, an *ftp* based throughput measure was developed. All of the tools are written in *sh* and *gawk*.

Diagnostic Reports

The goal of **pingit** is to report on network connectivity. The script *pingit.sh* uses *ping* statistics and *rcp* to determine the connectivity of the various segments of the NAS LAN. Because the goal is to report on subnet connectivity rather than host availability, at least two hosts on each subnet are chosen as sources and at least two hosts on each network or subnet being monitored are chosen as destinations. In addition, all interfaces of internal gateways and at least one interface of each of the NAS external gateways are monitored. This is crucial because the NAS LAN has many redundant paths and a gateway could stop functioning without having any impact on users.

The start times of *pingit.sh* are staggered in the crontab entries so that hosts are not bombarded with pings from many sources at once. By staggering the times, no host receives more than one set of 50 pings per minute from any of the source machines. Fifty pings per destination was chosen as a tradeoff between reasonable execution time and a large enough sample size for the packet loss statistics to be meaningful. With about 30 *ping* destinations, *pingit.sh* takes about 40 minutes to run.

In order to provide diagnostic information, the data produced by *pingit.sh* is processed by two variants of the same script, *watchit.sh* and *watchem.sh*. *Watchit.sh* is designed to be run on some of the machines where the data is collected and to provide immediate notification of failure. It runs on the sources which have the earliest and latest *pingit.sh* crontab entries. These machines were chosen for immediate reporting for the following reasons: the first *pingit.sh* to complete will hopefully be the one which was started first. So if there is a problem, the quickest notification will come from reporting on that data. If a problem were to develop after the first *pingit.sh* to run had completed, the last one of the hour has the best chance of catching the problem.

Watchit.sh reports on any pings which resulted in packet loss of 25 out of 50 packets or more. It also reports on any *rcp* failures. The source, destination, time and percentage of packet loss or *rcp* failure are reported in a tabular format. The report is mailed to everyone on the *net-watchers* mailing list [See Figure 1]. This reporting mechanism can provide notification of an outage within about 40 minutes of its occurrence.

In addition to the *watchit* report, there is a designated central machine to which all the source hosts copy their data. After all the collection for the hour is complete, *watchem.sh* compiles a report based on the information from all the sources. It uses the same tabular format as *watchit.sh*, but it will report on losses of 4 out of 50 packets or more. In addition it computes the hourly network metric numbers for the local, remote and gateway portions of the network. Because it cannot run until all the data has arrived, it is produced about 30 minutes after the other reports, so its information is not as timely [See Figure 2].

If a single host or segment of the network is down, each source host will report the event independently. This can provide important information for determining what portion of the network is down. In practice, in the NAS environment, it is more frequent to have a host or remote link down than to experience partial connectivity. The report could be made more readable by replacing the multiple lines reporting partial failure with a single line reporting probable complete failure, where appropriate.

Watchit.sh is also designed to run on a source host if it is unable to copy its data to the central machine. The assumption here is that if the data cannot be copied, there may be a problem with connectivity to the subnet of the central machine, and that needs to be made known.

The hourly reports produced by *watchit.sh* and *watchem.sh* are designed to notify staff of network problems relatively quickly. They provide *ping* statistics between subnets, to remote networks, to crucial resources on the network and to gateways which may fail invisibly otherwise. The reports have identified several cases where one of the redundant routers between NAS and the outside world had been kicked loose from its ethernet connector. The reports have also helped in troubleshooting problems with the ethernet processor board in the ethernet to Hyperchannel router and in finding a Hyperchannel

```
Date: Thu, 2 Aug 90 18:38:22 -0700
From: Network Administration
To: net-watchers
Subject: Network Trouble (watchit) Report  Thu Aug  2 18:38:21 PDT 1990
         Source             Destination  Loss           Time
       reynolds   jesgate.nas.nasa.gov    96% (48/50) 08/02 18:19
       reynolds          128.157.212.3  100% (50/50) 08/02 18:20
       reynolds           128.157.5.56    92% (46/50) 08/02 18:21
```

Figure 1 – Hourly mail from *watchit.sh*.

configuration problem with one of the NAS Crays.

A Network Metric

The network metric developed is a measure of network availability. With a valuable resource like a Cray Y-MP or Cray-2, one becomes very interested in knowing how many hours of the week such a resource is actually available to users. By analogy, this network metric measures the availability of the local and remote portions of the NAS network and is run on a weekly basis.

In order to measure network availability, one needs to define it. The network is entirely available if connections from any part of the network to any other part are possible. If it is impossible to make a connection from one subnet to another subnet, then that path is unavailable.

The network metric is broken into two components: local and remote. Since there are 5 local subnets, there are 4 + 3 + 2 + 1 = 10 possible local paths. There are 4 remote paths: three NAS links to other NASA sites and the connection between NAS and the rest of Ames (and hence the outside world). The remote metric only measures a fraction of the NAS long haul network, although the connections which are checked account for the majority of NAS long haul traffic. It is currently not possible to include the other links as they are dialup and often intentionally unavailable.

The local or remote availability for each hour is computed as a percentage of local or remote available paths to the total local or remote paths. To compute the availability, the data generated by *pingit.sh* is used. This data consists of pings and rcps between various hosts. Each hostname is replaced by the name of the subnet or network it is homed on, to eliminate dependence on any particular host. The pings are assigned a Boolean value, from which the availability of each path is readily computed. From that the local and remote network availability can be determined. This method of computation requires that a path must be down for a sustained

```
Date: Thu, 2 Aug 90 19:55:38 -0700
From: Network Administration
To: net-watchers
Subject: Network Summary Report  Thu Aug  2 19:55:36 PDT 1990
      Source               Destination  Loss          Time
        fs01      jesgate.nas.nasa.gov  100% (50/50) 08/02 19:06
        fs01             128.157.212.3   98% (49/50) 08/02 19:07
        fs01               128.157.5.56   94% (47/50) 08/02 19:07
        fs01    reynolds0.nas.nasa.gov  100% (50/50) 08/02 19:21
        fs01                 reynolds0  rcp  failed 08/02 19:23
        fs01    reynolds1.nas.nasa.gov  100% (50/50) 08/02 19:23
        fs02      jesgate.nas.nasa.gov   94% (47/50) 08/02 19:07
        fs02             128.157.212.3   94% (47/50) 08/02 19:08
        fs02               128.157.5.56   98% (49/50) 08/02 19:09
        fs02    reynolds0.nas.nasa.gov  100% (50/50) 08/02 19:22
        fs02                 reynolds0  rcp  failed 08/02 19:25
        fs02    reynolds1.nas.nasa.gov  100% (50/50) 08/02 19:25
                      /* data removed for brevity */
      navier      jesgate.nas.nasa.gov   94% (47/50) 08/02 19:17
      navier             128.157.212.3   88% (44/50) 08/02 19:18
      navier               128.157.5.56   96% (48/50) 08/02 19:18
      navier    reynolds0.nas.nasa.gov  100% (50/50) 08/02 19:31
      navier                 reynolds0  rcp  failed 08/02 19:33
      navier    reynolds1.nas.nasa.gov  100% (50/50) 08/02 19:33
  prandtl-ec      jesgate.nas.nasa.gov   92% (46/50) 08/02 19:02
  prandtl-ec             128.157.212.3   96% (48/50) 08/02 19:03
  prandtl-ec               128.157.5.56  100% (50/50) 08/02 19:03
  prandtl-hy    reynolds0.nas.nasa.gov  100% (50/50) 08/02 19:16
  prandtl-hy                 reynolds0  rcp  failed 08/02 19:18
  prandtl-hy    reynolds1.nas.nasa.gov  100% (50/50) 08/02 19:18
Network Availability for 1900 is 100 100 96.00 (local remote gateway)
```

Figure 2 – Hourly mail from *watchem.sh*.

amount of time, potentially as much as an hour, for it to be counted as unavailable in the network metric.

Each week, graphs of the local and remote availability are created, and local and remote weekly availability numbers are computed. The weekly availability is calculated as the area under the availability curve, normalized to a percentage. The weekly availability numbers are reported as part of the graphs [See Figure 3].

On a network where segments are rarely isolated from each other, this network metric is not very exciting. However, on a network where connectivity between segments is sporadic, this information is interesting and useful.

Measuring Throughput

To measure the throughput that scientists experience on the NAS local network, the scripts *wrapper*, *ftpscript* and *process.sh* were developed. Ftp is used to measure throughput because it reports statistics on the speed of the file transfer and it is a utility in common use. The data paths between NAS subnets and the Crays are focused on because they are the most important paths for the local NAS users.

The script *wrapper* runs every four hours and calls *ftpscript* to perform *ftp* transfers of a 1.4 megabyte binary file from two fileservers on each subnet to */dev/null* on each of the Crays, and from each Cray to */dev/null* on each of the fileservers. Daily, *process.sh* creates and mails a report which gives the throughput in kilobytes per second for each direction of the transfers between each of the subnets and each of the Crays [See Figure 4].

Throughput between the Crays and local subnets varies between 50 and 400 kilobytes per second. It is faster to transfer data from the Crays to the subnets than vice versa, as a rule. There is one subnet which has consistently poor performance relative to the other four. The problem is suspected to be internal to the subnet, but it has not been traced yet. The other subnets regularly experience throughput of over 300 kilobytes per second. The data is traveling over an ethernet, across a router, and over the Hyperchannel.

It is simple to modify the scripts to test throughput across other paths. It was of interest to compare the *ftp* throughput within three subnets: the subnet with extraordinarily slow throughput times to the Crays, the least crowded subnet and the most crowded subnet. Therefore, additional reports were generated to provide that information. The reports confirmed the hypothesis that the problem with throughput between the Crays and the one subnet is internal to that subnet, as the internal throughput on that subnet was far worse than even the heavily crowded subnet. The least crowded subnet has only 40 hosts, where the most crowded has over 200. As was expected, throughput on the crowded subnet drops to around 150 kilobytes/sec during the day, where on the uncrowded subnet, throughput of around 350 kilobytes/sec is common. These results were as expected, and have helped in accelerating

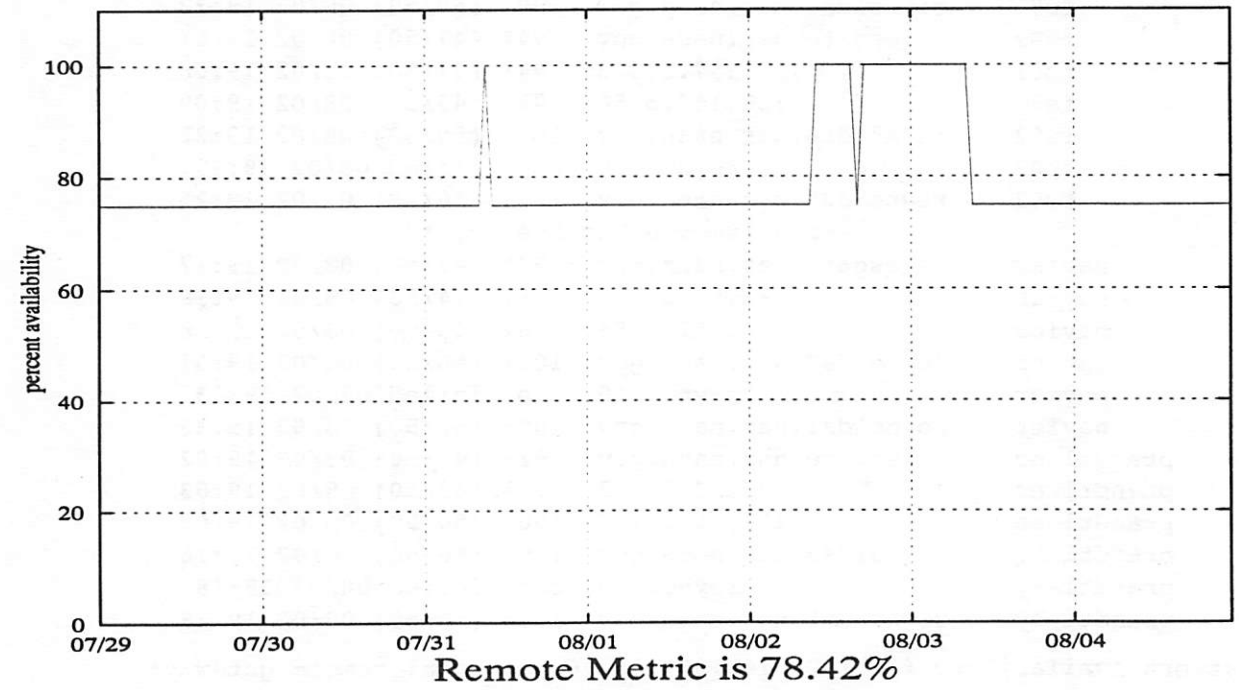

Figure 3 – Remote Network Availability from 07/29/90 to 08/04/90

plans for splitting up the largest subnet.

Conclusion

The network tools which were presented are in daily use at the NAS. The hourly report has proven to be a good tool for proactively responding to network problems that arise. The network metric has answered a management need to measure the uptime of the network and will become an interesting historical measurement as the NAS network grows more complex. The throughput times give a sense of what performance a user can expect. It will also provide something to compare against as the underlying network media becomes faster. The *ping* and *ftp* data are being archived so that historical studies are possible. Further analysis of this data will form a basis for evaluating the changes being made to the NAS network.

Reach the author at Computer Sciences Corporation; NASA Ames Research Center; M/S 258-6; Moffett Field, California 94035-1000; +1 415 604 4622 or at ktk@nas.nasa.gov electronically.

```
Date: Mon, 30 Jul 90 21:21:41 -0700
From: Network Administration
To: net-watchers
Subject: ftp report for 073090

FTP TRANSFER RATES FOR navier (kilobytes/second):
from Monday July 30, 1990 00:20 to Monday July 30, 1990 20:20

 dir   net    best   worst    0020    0420    0820    1220    1620    2020
  to net23     219      33     182     214     219     200     151      33
from net23      89      75      84      86      84      75      86      89

  to net32     341     129     301     289     309     341     294     129
from net32     343     186     280     251     186     292     295     343

  to net48     397     300     381     397     369     392     300     369
from net48     356     227     356     319     257     336     227     356

  to net64     243     132     243     235     185     216     132     204
from net64      91      69      69      90      76      91      69      73

FTP TRANSFER RATES FOR reynolds (kilobytes/second):
from Monday July 30, 1990 00:20 to Monday July 30, 1990 20:20

 dir   net    best   worst    0020    0420    0820    1220    1620    2020
  to net23     163     140      84     140     140     163     141       0
from net23     101      68      91      94      89      84      68     101

  to net32     306     217     103     268     217     271     306       0
from net32     334     168     264     227     168     297     254     334

  to net48     358     232     127     332     330     358     232       0
from net48     352     222     352     303     222     288     254       0

  to net64     222      98     165     222     188     217      98       0
from net64     169     113     169     132     126     113     118       0
```

Figure 4 – Daily Throughput Report

LISA IV
October 17-19, 1990
Colo. Springs, Colorado

Using *expect* to Automate System Administration Tasks

Don Libes – National Institute of Standards and Technology

ABSTRACT

UNIX system administration often involves programs designed only for interactive use. Many such programs (**passwd**, **su**, etc.) cannot be placed into shell scripts. Some programs (**fsck**, **dump**, etc.) are not specifically interactive, but have poor support for automated use.

expect is a program which can "talk" to interactive programs. A script is used to guide the dialogue. Scripts are written in a high-level language and provide flexibility for arbitrarily complex dialogues. By writing an **expect** script, one can run interactive programs non-interactively.

Shell scripts are incapable of managing these system administration tasks, but **expect** scripts can control them and many others. Tasks requiring a person dedicated to interactively responding to badly written programs, can be automated. In a large environment, the time and aggravation saved is immense.

expect is similar in style to the shell, and can easily be mastered by any system administrator who can program in the shell already. This paper presents real examples of using **expect** to automate system administration tasks such as **passwd** and **fsck**. Also discussed are a number of other system administration tasks that can be automated.

Keywords: **expect**, **fsck**, interaction, **passwd**, password, programmed dialogue, security, shell, Tcl, UNIX, **uucp**

Introduction

UNIX system administration often involves using programs designed for interactive use. Many such programs (**passwd**, **su**, etc.) cannot be placed into shell scripts. Some programs (**fsck**, **dump**, etc.) are not specifically interactive, but have little support for automated use.

For example the **passwd** command prompts the user for a password. There is no way to supply the password on the command line. If you use **passwd** from a shell script, it will block the script from running while it prompts the user who invoked the shell script.

Because of this, you cannot, for example, reject passwords that are found in the system dictionary, a common security measure. It is ironic that security was the reason that **passwd** was designed to read directly from the keyboard to begin with.

passwd is not alone in this recalcitrant behavior. Many other programs do not work well inside of shell scripts and quite a few of these are crucial tools to the system administrator. Examples are **rlogin**, **telnet**, **crypt**, **su**, **dump**, **adb**, and **fsck**. More problems will be mentioned later.

The problem with all of these programs is not the programs themselves, but the shell. For example, the shell cannot see prompts from interactive programs nor can it see error messages. The shell cannot deal with interactive programs this way because it is incapable of creating a two-way connection to a process. This is an inherent limitation of classic UNIX shells such as **sh**, **csh** and **ksh** (from here on generically referred to as simply *the shell*).

expect – An Overview

expect is a program that solves the general problem of automating interactive programs. **expect** communicates with processes by interposing itself between processes (see Figure 1). Pseudo-ttys are used so that processes believe they are talking to a real user. A high-level script enables handling of varied behavior. The script offers job control so that multiple programs can be controlled simultaneously and affect one another. Also, a real user may take and return control from and to the script whenever necessary.

expect is a general-purpose system for solving the interactive program problem, however it solves an unusually large number of problems in the system administration arena. While the *UNIX style* is to build small programs that can be used as building blocks in the construction of other programs using shells and pipelines, few system administration programs behave this way.

Traditionally, little time was spent designing good user interfaces for system administrator tools. The reasons may be any or all of the following:

- System administrators were experienced programmers, and therefore didn't need all the hand-holding that general user programs require.
- Programs such as **fsck** and **crash** were run infrequently, so there was little point spending much time on such rarely used tools.
- System administration tools were used in extreme conditions, considered not worth programming for because of their difficulty or rarity. It was more cost-effective to solve the problem by hand in real-time.
- System administrators solved problems in site-dependent ways, never expecting their underdesigned programs to be propagated widely.

Whatever the reason, the result is that the UNIX system administrator's toolbox is filled with representatives of some of the worst user interfaces ever seen. While only a complete redesign will help all of these problems, **expect** can be used to address a great many of them.

Example – passwd

The **expect** script in Listing 1 takes a password as an argument, and can be run non-interactively such as by a shell script. A shell script could prompt and reject easily guessed passwords. Alternatively, the shell script could call a password generator. Such a combination could create large numbers of accounts at a time without the system administrator having to hand-enter passwords as is currently done.

Admittedly, the script reopens the original security problem that **passwd** was designed to solve. This can be closed in a number of ways. For example, **expect** could generate the passwords itself by directly calling the password generator from within the script.

The scripting language of **expect** is defined completely by Libes [1][2] and Ousterhout [3][4]. In this paper, commands will be described as they are encountered. Rather than giving comprehensive explanations of each command, only enough to understand the examples will be supplied.

```
set password [index $argv 2]
spawn passwd [index $argv 1]
expect {*password:}
send $password\r
expect {*password:}
send $password\r
expect eof
```

Listing 1 – Non-interactive **passwd** script. First argument is username. Second argument is new password.

set – Sets the first argument to the second (i.e., assignment).

In line 1 of the script, the first argument to **set** is **password**. The second is an expression that is evaluated to return the second argument of the script by using the **index** command. The first argument of **index** is a list, from which it retrieves the element corresponding to the position of the second argument. **argv** refers to the arguments of the script, in the same style as the C language **argv**.

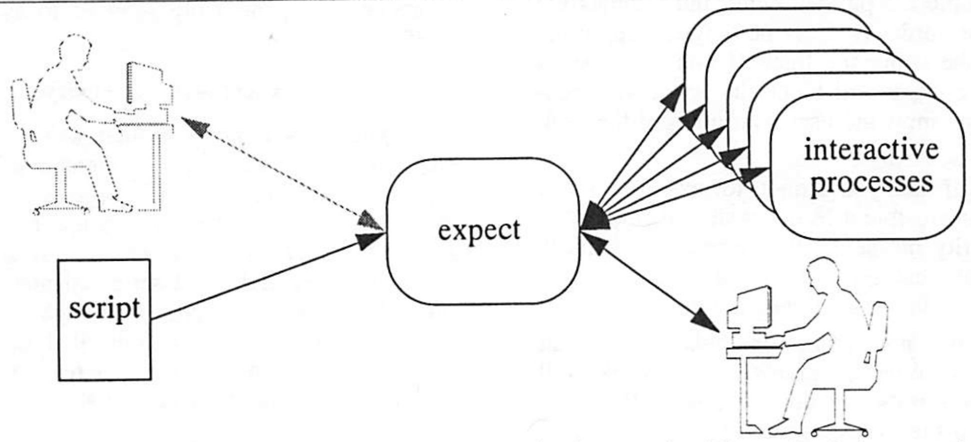

Figure 1 – **expect** is communicating with 5 processes simultaneously. The script is in control and has disabled logging to the user. The user only sees what the script says to send and is essentially treated as just another process.

spawn – Runs an interactive program.

The spawned program is referred to as the *current process.* In this example, **passwd** is spawned and becomes the current process. A username is passed as an argument to **passwd.**

expect – Looks for a pattern in the output of the current process.

The argument defines the pattern. Additional optional arguments provide alternative patterns and actions to execute when a pattern is seen. (An example will be shown later.)

In this example, **expect** looks for the pattern `password`. The asterisk allows it to match other data in the input, and is a useful shortcut to avoid specifying everything in detail. There is no action specified, so the command just waits until the pattern is found before continuing.

send – Sends its arguments to the current process.

The password is sent to the current process. The `\r` indicates a carriage-return. (All the "usual" C conventions are supported.) There are two **send/expect** sequences because **passwd** asks the password to be typed twice as a spelling verification. There is no point to this in a non-interactive **passwd,** but the script has to do this because **passwd** doesn't know better.

The final `expect eof` searches for an end-of-file in the output of **passwd** and demonstrates the use of *keyword patterns.* Another one is **timeout,** used to denote the failure of any pattern to match. Here, eof is necessary only because **passwd** is carefully written to check that all of its I/O succeeds, including the final newline produced after the password has been entered a second time.

It is easy to add a call and test of `grep $password /usr/dict/words` to the script to check that a password doesn't appear in the on-line dictionary, however, we will leave the illustration of control structures to the next example.

Example – fsck

Many programs are *ostensibly* non-interactive. This is, they can run in the background but with a very reduced functionality. For example, **fsck** can be run from a shell script only with the **-y** or **-n** options. The manual [5] defines the **-y** option as follows:

"Assume a yes response to all questions asked by fsck; this should be used with extreme caution, as it is a free license to continue, even after severe problems are encountered."

The **-n** option has a similarly worthless meaning. This kind of interface is inexcusably bad, and yet many programs have the same style. For example, **ftp** has an option that disables interactive prompting so that it can be run from a script, but it provides no way to take alternative action should an error occur.

Using **expect,** you can write a script that allows **fsck** to be run, having questions answered automatically. Listing 2 is a script that can run **fsck** unattended while providing the same flexibility as being run interactively. The script begins by spawning **fsck.**

for – Controls iteration (looping).

The language used by **expect** supports common high-level control structures such as **if/then/else.** In the second line, a **for** loop is used which is structured similarly to the C-language version. The body of the **for** contains one **expect** command.

```
spawn fsck
for {} 1 {} {
    expect eof                              break \
           {*UNREF\ FILE*CLEAR?\ }          {send y\r} \
           {*BAD\ INODE*FIX?\ }             {send y\r} \
           {*?\ }                           {send n\r}
}
```

Listing 2 – Non-interactive **fsck** script.

```
spawn fsck
for {} 1 {} {
    expect eof                                   break \
           {*UNREF\ FILE*CLEAR?\ }               {send n\r} \
           {*BLK(S)\ MISSING*SALVAGE?\ }         {send y\r} \
           {*?\ }                                {interact +}
}
```

Listing 3 – User-friendly **fsck** script.

The following **expect** command demonstrates the ability to look for multiple patterns simultaneously. (The backslashes (\) are used to quote characters – in this case whitespace.) In addition, each pattern can have an accompanying action to execute if the pattern is found. This allows us to prespecify answers for specific questions. When the questions `UNREF FILE...CLEAR?` or `BAD INODE NUMBER...FIX?` appear, the script will automatically answer `y`. If anything else appears, the script will answer n.

In general, if all questions are known and answerable in advance, a script can be run in the background. With more complex programs it may be desirable to trap unexpected questions and force a user to interactively evaluate them. Listing 3 is a script does exactly this.

If the script does not match one of the prespecified answers, the last case (`{*?\ }`) matches. (The ? is necessary to prevent the script from triggering before the entire question arrives.) The **interact** action passes control from the script to the keyboard (actually **stdin**) so that a human can answer the question.

interact – Pass control from script to user and back.

During **interact**, the user takes control for direct interactions. Control is returned to the script after pressing the optional escape character. In this script, + is chosen as the escape character by passing it as the argument to **interact**.

A real **expect** script for **fsck** would do several other things. For example, **fsck** uses several statically-sized tables. For this reason, **fsck** is limited to the number of errors of one type that can be fixed in a single pass. This may require **fsck** be run several times. While the manual says this, **fsck** doesn't, and few system administrators know **fsck** that intimately. When run from a shell script, this lack of programmability will cause the system to come up all the way with a corrupt file system (if the return code isn't checked) or be unnecessarily rebooted several times (if the return code is checked).

Example – Callback

The script in Listing 4 was written by a user who wanted to dial up the computer, and tell it to call him back. Since he lived out of the local calling area, this would get the computer to pick up his long-distance phone bills for him.

```
spawn tip modem
expect {*connected*}
send ATDT[index $argv 1]\r
set timeout 60
expect {*CONNECT*}
```

Listing 4 – Callback script. First argument is phone number.

The first line spawns **tip** which opens a connection to a modem. Next, **expect** waits for **tip** to say it is connected to the modem. The user's phone number, passed as the first argument to the script, is then fetched and added to a command to dial a Hayes-compatible modem. A carriage-return is appended to make it appear as if a user had typed the string, and the modem begins dialing.

The third line assigns 60 to the variable **timeout**. **expect** actually looks at this variable in order to tell it how many seconds to wait before giving up. Eventually the phone rings and the modem answers. **expect** finds what its looking for and exits. At this point **getty** wakes up, and finding that it has a dialup line with DTR on it, starts **login** which prompts the user to log in.

Since the script was originally written, we have added a few more lines to automate and verify phone numbers based on the uid running it partly for security, but the fragment shown here was used successfully and forms the heart of our current script. Ironically, we recently noticed a 60Kb equivalent to **callback** on Usenet that had no more functionality than a dozen or so lines of **expect**.

Of course, not all scripts are this short. I'm limited to what can be presented here, and these examples really serve just to give you a feel of what **expect** does and how it can be applied. What is

```
spawn ftp
. . .
send ls * lsFile\r             ;expect *success*ftp>*
set lsVar [exec cat lsFile]
exec rm lsFile\r
set len [length $lsVar]
for {set i 0} {$i < $len} {set i [expr $i+1]} {
    set file [index $lsVar $i]
    send get $file\r           ;expect *success*ftp>*
    send delete $file\r        ;expect *success*ftp>*
}
```

Listing 5 – Fragment of **ftp** spool script.

important is that **expect** scripts are small and simple for problems that are small and simple. **expect** obviates the need for resorting to C just because of limitations on the part of the shell.

Example – Intelligent ftp

One of our site administrators wanted to spool files in a directory. Later, a second computer would use **ftp** to pick them up and then delete them from the first computer. His first attempt was to use `mget *` followed by `mdelete *`. Unfortunately, this deletes files that arrive in the window between when the **mget** starts and the **mdelete** starts. The script fragment in Listing 5 solved the problem.

The script begins by spawning **ftp**. I have omitted several lines that open a connection followed by sending and confirming the user and password information. The next line sends an **ftp** command to store the list of remote files in a local file called **lsFile**. This command is terminated by a semicolon, allowing the response to be verified with an **expect** command on the same line of the script.

exec – Execute a UNIX command.

exec executes a UNIX command and simply waits for it to complete, just as if it were in a shell script. In line four, **cat** returns the list of files, and their names are stored in the variable, **lsVar**. **exec** is used again in the next line, this time to delete the local file, **lsFile**.

The remainder of the script merely iterates through the variable **lsVar**, sending **get** commands followed by **delete** commands for each file found in the earlier **ls**.

Other examples solved

expect addresses a surprisingly large class of system administration problems which before now have either been solved by avoidance or special kludges. At the same time, **expect** does not attempt to subsume functions already handled by other utilities. For example, there is no built-in file transfer capability, because **expect** can just call a program to do that. And while the shell is programmable, it cannot interact with other interactive processes and it cannot solve any of the examples in this paper.

In this section, more examples will be discussed. Because of space limitations, scripts will not be shown, but all of them have been written and are being used.

Regression testing

Testing new releases of interactive software (**tip**, **telnet**, etc.) requires a human to press keys and watch for correct responses. Doing this more than a few times becomes quite tiresome. Naturally, people are much less likely to run thorough regression tests after making small changes that they think probably don't affect other parts of a program.

Regression testing can also be useful for your entire installation. You can make a script that tests all your site's local applications, and run it at after each system upgrade or configuration change.

Automating logins

Many programs have a frequently repeated, well-defined set of commands and another set that are not well-defined. For example, a typical **telnet** session always begins with a log in, after which the user can do anything. To automate this, **expect** has the ability to pass control from the script to the user. At any time, the user can return control to the script temporarily to execute sequences of commonly repeated commands.

At my site, **expect** is heavily used to automate the process of logging in through multiple frontends and communication switches. In fact, the original reason **expect** was written was to create six windows, each of which automatically logged in to another host to run a demo.

The general idea of automating **telnet**, **ftp**, and **tip** is very useful when dealing with hosts that do not support **rlogin** and **rcp**. But the technique is also useful with native UNIX commands like **su**, **login**, or **rlogin**. **expect** scripts can call any of them, sending passwords as appropriate and then continuing actions as desired. While any of these commands can be embedded in a shell script, the shell has no way of taking control over what happens *inside* of these programs. Subsequent commands from the shell script do not get sent to the new context, but are held up until the previous command has completed so that they can be sent to the original context. **expect** has no such problems switching contexts to continue controlling any of these sessions.

telnet – It's not just for breakfast anymore

telnet also functions as an interface to the exciting world of TCP sockets. **telnet** can be used to access non-**telnet** sockets and query other hosts for their date (port 13), time (port 37), list of active users (port 11), user information (port 43), network status (port 15), and all sorts of other goodies that you might only be able to get if you had permission to log in.

For example, our site regularly runs a script that checks (port 25) what version of **sendmail.cf** each of our local hosts is actually using. If we did this by reading files, we would need permission to log in, or remotely mount file systems and read directories and files on several hundred hosts. Using **telnet** is much easier, albeit a little strange.

su, passwd, crypt and other password-eaters

Programs that read and write **/dev/tty** cannot be used from shell scripts without the shell script accessing **/dev/tty**. An earlier example showed how to force **passwd** not to read from **/dev/tty**. With this technique, you can change its input source to **stdin**, a parameter, or even an environment variable.

As another example, suppose you have typed a command that fails because you weren't root. The typical reaction is to type **su** and then reenter the command. Unfortunately, history won't work in this situation as `!-2` will just evoke the error `-1: Event not found`. The problem is that you want to refer to a command that is now in a different shell instantiation, and there is no way to get back to it.

A solution is to pass the failed command as an argument (via `!!`) to an **expect** script that will prompt you for the root password, invoke **su**, and then feed the original failed command to the resulting superuser shell. If the **expect** script executes **interact** as its last action, you will have the original command executed for you (no retyping), plus you will get a new superuser shell. There is no way to do this with **su** except by resorting to temporary files for your history and a lot of retyping.

A more painful example is **newgrp**. Unlike **su**, **newgrp** does not allow additional arguments on the command line to be passed to the new shell. You must interactively enter them after **newgrp** begins executing. In either case, both **su** and **newgrp** are essentially useless in shell scripts.

Security – The good news is ...

Earlier, I mentioned how to build a script that would force users to choose good passwords without rewriting **passwd**. All other alternatives either rewrite the **passwd** program or ask the user to be responsible for choosing a good password.

On the opposite side of the coin, **expect** can be used to test other sites for secure logins (or to break in, I suppose). Trying to login as **root** using, say, all the words in an on-line dictionary, at all the local hosts at a site would be prohibitively expensive for a human to do. **expect** would work at it relentlessly, eventually finding an insecure **root**, or showing the site to be protected by good passwords.

Questions at boot time

While booting, it is useful to validate important system facts (e.g., **date**) before coming up all the way. Of course, if no one is standing in front of the console (e.g., the system booted due to a power failure) the computer should come up anyway. Writing such a script using the shell is painful, primarily because a read-with-timeout is not directly implemented in the shell. In **expect**, all reads timeout. **expect** can prompt and read from the keyboard just as easily as from a process.

Transferring hierarchies with ftp

Anonymous **ftp** is very painful when it comes to directory hierarchies. Since there is no recursive copy command, you must explicitly do **cd**s and **get**s. You can automate this in a shell script, but only if the hierarchy is known in advance. **expect** can execute an **ls** and look at the results so that you can transfer a hierarchy no matter what it looks like or how deep it is. **expect** supports recursive procedures, making this task a short script. My site regularly retrieves large distributions (e.g., Gnu, X) this way.

Assisting adb and other "dumb" programs

Quite often, vendors provide instructions for modifying systems in the form of **adb** instructions, where some instructions may depend on the results of earlier ones (i.e., "*each time _maxusers is incremented, you must add 16 to _nfile*"). **adb** has no special scripting language that supports such interaction, nor does the shell provide this capability. **expect** can perform this interaction, playing the part of the user, by directly looking at the results of operations, just as a user would.

This technique can be applied to any program. In fact, **expect** can act as an intermediary between the user and programs with poorly-written user interfaces. **expect** normally shows the entire dialogue but can be told not to. Then **expect** can prompt the user for commands such as `show _maxusers` instead of **adb**'s native but cryptic `_maxusers/d`. Translations can also be performed in the reverse direction. A short **expect** script could limit the difficulty of system administrators who have no interest in mastering **adb**. In addition, the ability of system administrators to accidentally crash the system by a few errant keystrokes would be dramatically lessened.

Grepping monster log files

A common command sequence involves looking at a log with, say, **grep**, and then interrupting it (with ^C) after the line of interest appears. Unfortunately, **grep** and other programs are limited to the amount of programmability they have. For example, **grep** can not be directed to stop searching after the first match. A short **expect** script can send an interrupt to **grep** after seeing the first line just as if the user were actually at the keyboard.

With programs that generate log files as large as a gigabyte, this is a real problem. Without **expect**, the only solutions are to let **grep** continue running over the whole file, or to dedicate a human to the task of pressing ^C at the right time. **expect** can cut off the process as soon as possible, mailing the results back the system administrator if necessary.

In general, **expect** is useful for sending odd characters to a process that cannot be embedded in a shell script. **expect** can also execute job control commands (**bg**, **fg**, etc.) in order to mediate between processes that were never designed to communicate with each other. Again, this can relieve a human from the tedious task of interactively monitoring programs.

Administering non-UNIX systems

expect is a UNIX program, yet it can be used to administer non-UNIX systems. How is this possible? Running **telnet** (**tip**, **kermit**, etc.) to a non-UNIX host, it can log in and perform **send/expect** sequences on the remote computer. The operating system or environment of the remote computer is completely irrelevant to **expect**, since all of this is isolated to the **expect** script itself.

This is very useful for system administrators that already have a UNIX computer on their desk but are forced by management to administer another computer. ("*You already administer 20 UNIX systems. How much more work could it possibly be for you to administer just one more system? Oh, and it runs VMS.*")

Security

Several of the examples presented have prompted for passwords that are different than the usual UNIX style. Normally, UNIX prompts for passwords directly from **/dev/tty**. This has the unfortunate drawback that you cannot redirect **stdin**. We have shown how to get around that by using **expect**.

Of course, doing this reopens a possible security hole. Unprivileged users can detect passwords passed as arguments by using **ps**. If passwords are stored in files, lapses in security can make plaintext passwords evident to people browsing through your files. Publicly-readable backup media are one of the simplest such security lapses.

If you are at all interested in security, I do not recommend storing plaintext passwords in files. The likelihood of such a password being discovered and abused is just too high. Our users store passwords in files, but only for highly restricted accounts, such as for demos or anonymous **ftp**.

The chances of leaking a password through **ps** are lower, and can be lowered further still by using the smallest possible script around the password prompting program. Such a window is extremely small. Nonetheless, secure sites should not take even this chance.

An alternative is to have **expect** interactively prompt for passwords. If you have an **expect** script that is doing a complicated series of **telnets**, **ftps** and other things, the scripts can encode everything but the passwords. Upon running such a script, the user will be only be prompted once for a password, and nothing else. Then **expect** will use that password whenever necessary, and complete all the other dialogue from data pre-stored in files.

In summary, **expect** need not weaken security. Used wisely, **expect** can even enhance security. However, you must use common sense when writing scripts.

Comparison to other system administration tools

This section of the paper can be considered controversy or heresy, as you wish. It is somewhat religious in that the arguments can only be resolved by philosophical choice rather than logic. I have kept it down to a very few reasons to give you only the barest feelings for what I consider is important to understand when choosing **expect** over other system administration tools.

As should be obvious, I think there are very few alternatives to using **expect**. Traditionally, the popular choices have been 1) avoidance and 2) C programming. These are now no longer the only choices.

Shell

The shell is incapable of controlling interactive processes in the way that **expect** can. Nonetheless, certain comparisons between **expect** and the shell are inevitable. In particular, **expect** includes a high-level language that is interpreted and bears a strong similarity to the shell and also to C. In that sense, I see little to argue about since **expect** can do shell-like functions. In a previous paper [1], I have suggested the addition of **expect**'s features to the shell. No one wants to learn yet another shell, and there is no reason why these capabilities cannot be added to the shell.

Perl

A more interesting comparison is with Perl, a language claimed (by the author) [6] to embody the best aspects of the shell, C, **awk**, **sed**, and a number of other UNIX tools. Having spent some time programming in Perl, there is no question in my mind that Perl is capable of solving the same tasks that I have described in this paper. Pseudo-tty packages for Perl have been written and **send/expect** utilities could be written also.

Perl is a very powerful language. It is much richer than the language used by **expect** (or any shell for that matter). This has advantages and disadvantages. The most obvious disadvantage is that Perl's overabundance of options and features simply aren't necessary for the tasks that **expect** addresses. Perl's complexity is reflected in its disk space. The computer on my desk, a Sun 3, requires 270K to store Perl and has a significant startup time. **expect**, on the other hand, is 70K with essentially no startup time. There are other reasons that Perl is not widely applied to certain problems, but completing the discussion deserves a paper of its own.

Instead I will summarize by saying that **expect** is appropriate to only a fraction of the system administration problems that Perl solves. This is intentionally so. **expect** was written to solve a very specific problem, and it does that concisely and efficiently. I think that it fits well with the UNIX philosophy of small tools, unlike Perl which demands a significant investment in mastering its complexity. Given the choice, I predict that most system administrators would choose a tool like **expect** that takes very little effort to learn, rather than entering the world of Perl.

Emacs

Emacs is analogous to Perl in many ways, including its flexibility and overabundance of functionality. Similarly, Emacs can be used to solve these same problems. And for much the same reasons as I gave above, Emacs is inappropriate for the class of problems I have suggested in this paper. Indeed, considering that Emacs has been available for over a decade, and I've never heard of anyone using it this way, I'll proffer that Emacs is so inappropriate for these problems, that it is not surprising this usage has never even occurred to anyone.

Conclusion

UNIX shells are incapable of controlling interactive processes. This has been at the root of many difficulties automating system administration tasks. While the UNIX community is gradually providing better designed tools and user interfaces, even more programs are being written with embarrassingly poor user interfaces at the same time. This is understandable because system administrators give more priority to solving a problem so they can go to the next one, than going back to pretty up an old and working solution.

expect is designed to work with programs as they are. Programs need not be changed or redesigned, no matter how poorly written. Understandably, the majority of system administrators are reluctant to modify a program that works and that they have not written themselves. Most prefer writing shell scripts using the classic UNIX tools philosophy.

expect handles these problems, solving them directly and with elegance. **expect** scripts are small and simple for problems that are small and simple. While not all **expect** scripts are small, the scripts scale well. They are comparable in style to shell scripts, being task-oriented, and provide synergy with shell scripts, both because they can call shell scripts and be called by them. Used judiciously, **expect** is a welcome new tool to the workbench of all UNIX system administrators.

Acknowledgments

This work was jointly funded by the NIST Automated Manufacturing Research Facility (AMRF, project 734-3385) and Scientific and Technical Research Services (STRS, project 734-3106).

The **callback** script was written by Scott Paisley. Walter Rowe wrote the **sendmail** version-checking script mentioned in the paper. Thanks to Walter Rowe and Ken Manheimer for their helpful comments on this paper. Sue Mulroney was helpful in correcting my badly grammar.

Availability

Since the design and implementation of **expect** was paid for by the U.S. government, it is in the public domain. However, the author and NIST would like credit if this program, documentation or portions of them are used. **expect** may be **ftp**'d as **pub/expect.shar.Z** from **durer.cme.nist.gov**. **expect** will be mailed to you, if you send the mail message `send pub/expect.shar.Z` to **library@durer.cme.nist.gov**.

References

[1] Don Libes, "**expect**: *Curing Those Uncontrollable Fits of Interaction*", Proceedings of the Summer 1990 USENIX Conference, Anaheim, CA, June 10-15, 1990.

[2] Don Libes, "*The* **expect** *User Manual – programmatic dialogue with interactive programs*", NIST IR 90-X, National Institute of Standards and Technology, November, 1990.

[3] John Ousterhout, "*Tcl: An Embeddable Command Language*", Proceedings of the Winter 1990 USENIX Conference, Washington, D.C., January 22-26, 1990.

[4] John Ousterhout, "*tcl(3) – overview of tool command language facilities*", unpublished manual page, University of California at Berkeley, January 1990.

[5] AT&T, UNIX Programmer's Manual, Section 8.

[6] Larry Wall, "*Perl – Practical Extraction and Report Language*", unpublished manual page, March 1990.

Don Libes received a B.A. in Mathematics from Rutgers University and an M.S. in Computer Science from the University of Rochester. Currently at the National Institute of Standards and Technology, Don is engaged in research that will help U.S. industry measure the standard hack. Unfortunately, NIST does not have a very good sense of humor, so he was forced to write his first book "Life With UNIX" through Prentice-Hall.

LISA IV
October 17-19, 1990
Colo. Springs, Colorado

Policy as a System Administration Tool

Elizabeth D. Zwicky – SRI International
Steve Simmons and Ron Dalton – Industrial Technology Institute

ABSTRACT

All decisions about how to manage a given system are made with respect to local policy. This is true even in the absence of such policy, as the consistent actions of the system manager become de facto policy [Hovell]. This paper will discuss the interactions between policy and systems management. Using a series of case studies, we will illustrate two points: how proper policies can be used to ease the day-to-day tasks of systems administration; and how technical issues can and should be used as one of the driving forces in policy decisions.

Introduction

At first glance policy is a political issue rather than a technical issue. But policy made without regard to technical issues is a recipe for organizational and administrative disaster. Conversely, letting technical considerations dictate policy is a recipe for political disaster.

Policies are often a major factor in technical decisions; for instance, a backup system cannot be satisfactorily designed without an existing policy about what files get backed up by whom how often. Ohio State University's Computer and Information Science department (OSU-CIS) and SRI International's Information, Telecommunication and Automation Division (for brevity, called "SRI" throughout this document) back up roughly equivalent amounts of disk space, on the same types of machines, to the same sorts of tape drives, using programs written in the same language, and containing some of the same code – but the programs are completely different, because they implement very different policies.

To further complicate the situation, most systems are administered in a policy vacuum. The administrator may set de facto policies, but rarely will there be any formal recognition of those policies by management. This may at first seem depressing, but used properly can be a method of easing system administration.

This paper will discuss a number of policies both formal and informal, the technical and administrative needs driving those policies, and the results of their application. Unless otherwise stated, no names have been changed to protect the innocent. The guilty are left anonymous.

Points of contention

Policies are developed primarily because of conflicts between users and system administrators. Most system administrators have a sense that certain things are common sense (for instance, it seems intuitive to most that a single user should not have multiple accounts on the same system). It comes as a blow to discover that users do not usually share these intuitions. The following sections discuss some common points of conflict in large installations.

Independence vs. Service

Users often want or need to do eccentric things with their machines. By contrast, in order to make them more easily managed the system administrators prefer that all the machines be as close to identical as possible. Policies in this area serve two purposes; they provide polite and relatively unarguable ways in which to say "Not on my network you don't", and they provide clear statements of the price a user must pay for independence. In general, they have to give up something in order to have full control, and they are made deeply miserable if they have their own control and do something stupid. Users need to have a good idea exactly what they give up and exactly what the consequences are.

Some relevant questions:

- Do users get root on their machines?
- Do they get disks on them?
- Do they get to modify the system software?
- Do they get to run other operating systems?
- Can they inflict any sort of machine they take a fancy to on you?
- If they do any of this, how do they get backups, operating system upgrades, mail, news, access to printers, access to networks?
- Who decides what the machines are named?

Case Study: The Untrustworthy Hosts

At the Industrial Technology Institute (ITI) there was (and still is) a fairly large laboratory devoted to implementing MAP/TOP protocols on UNIX systems. This required a great deal of device driver work, kernel builds and installs, and kernel-level debugging. This was not a problem for the systems administration staff because the researchers gladly gave up central support in turn for unlimited root access (they later came to regret that, but that's another paper). Relations between the two groups were reasonably cordial once areas of responsibility and authority had been decided. The administrative staff did the backups and co-ordinated hardware and software maintenance projects; the researchers added and deleted accounts, managed their own disk usage, etc.

Problems began once the project was completed. The systems had been purchased by the research group with research funds. They were on a private ethernet, not connected to the central network. The researchers were not anxious to give up their windowed dedicated development environment, particularly when this meant returning to ASCII terminals on an overloaded VAX 785. They wanted to connect their systems to the central network and work from the lab. They wanted to send and receive mail, read news, have access to the Internet, mount NFS partitions, and all the goodies one expects in a well-connected environment. But they refused to give up root access, justifiably citing the ongoing support tasks of the original project.

Describing lab systems as "insecure" would be an understatement. Many accounts did not have passwords; other accounts existed for users who had left years before. Given we had been at least brushed by previous break-in attempts and the Morris worm, there was a great deal of resistance to allowing unlimited connectivity.

The policy decision made was largely driven by technical issues, and with very little management involvement. The issues:

External Security

It was decided to attach lab machines to the network, but not provide any external routing (we use static routing internally, including our gateway). This permitted the lab machines to be attached but not be accessible from (or to) the Internet. This effectively removed the issue of mail – they could do it only if they developed the sendmail expertise to forward everything to a trusted host and faked the return addresses via a hiddennet. News works fine through NNTP, and in this case the service was carefully configured to hide the laboratory hosts.

Internal Security

Entries in host tables and domain name service were made to identify the hosts on the trusted network, but those hosts were not placed in **/etc/hosts.equiv** on the trusted systems. Thus they could not **rlogin**, **rsh**, etc, without supplying a password. While this was an inconvenience, most users quickly came to terms with it. Now that the users have discovered **.rhosts** files (and use them in spite of requests not to) the appropriate changes are being made to no longer allow **.rhosts** to override **hosts.equiv**. For a broader solution to the same problem, see [Harrison].

Improved Security

The users still desire NFS mounts, access to Internet, etc. They were understanding of the security needs, and requested a technical solution that would permit it. We proposed and they accepted the use of *cops* [Farmer] as a security check to validate their security. When all their systems pass a *cops* audit, they will be added to the trusted hosts.

Results and Re-Evaluation

In this case purely technical issues drove the creation of a policy. In every request we were able to provide both a technical reason for a policy and technical means that would permit modification of the policy (In retrospect it was actually a benefit to have been touched by previous security problems -- they convinced the user community that security was a real issue.)

This policy has eased the integration of new computers into our network. As workstations and PC-based UNIXes appear on desktops, the policy developed for laboratory machines has been extended to apply to desktop systems. Having a policy in place made it much simpler to deal with objections. In the case of users who wished to fight the policy, we invited them to form a committee and make a policy acceptable to all. This being an impossible task, the users have thus far yielded to the inevitable.

Case Study: SRI

Because of SRI's somewhat baroque financial arrangements, it is very clear which machines are and are not maintained by the staff; if we charge you an hourly fee to use your machine, it's a facility machine. You can, of course, refuse to pay us, in which case the machine is your own; we can also refuse to accept your machine as a part of the facility, if it is not like the rest of our machines. If you do not pay an hourly fee, you pay on a time and materials basis, for a minimum of half an hour, every time we do anything at all for your machine.

For practical reasons we need to offer some services to non-facility machines. (We own all of the networks and all of the printers.) On the other hand, the money we charge pays our salaries; we can't afford to offer all of our services to people who aren't paying us. Furthermore, it is unfortunately easy for poor configuration on a non-facility machine to make life unlivable for facility machines.

Our compromise has been to allow non-facility machines to connect to the network, charging only for required hardware, and to register them in our name servers. In return, they are required to register all networks and hosts with us, and to configure their machines so as not to interfere with network operations. Hosts that are not well-behaved are disconnected from the ethernet, without any particular attempt at kindness. Hosts that can manage either to speak directly to our Ethernet-based Imagens, or to speak to a Berkeley line printer daemon, get printer access (in the latter case, via a special printer equivalence file, not hosts.equiv).

Other services are available at an hourly charge for the time we spend providing them, with other restrictions as needed. For instance, we will provide backups for hosts; we require control of root privileges on machines that we need to trust for this purpose, we charge for the hours required to set the system up (on a modern Sun running a modern SunOS, this is our minimum half-hour charge; on other machines it may run to 20 or 40 hours, especially if they are non-UNIX machines for which we have to devise new backup systems), and we charge on a weekly basis for the labor involved in running and monitoring the backups (usually half an hour a week). We do not attempt to charge for media, and we do not charge for restores, so long as they are infrequent.

The result is that there is usually considerable monetary advantage to a project in turning over machines to us if we are willing to take them. The hourly fee works out to much less than people normally ending up paying us for assistance, especially if they want to be reasonably integrated with the rest of the division. For machines that are capable of using facility services like NIS (previously YP) service, NFS mounting of file systems, and so on, there is an uncomfortably large grey area. Hosts that we trust because of backups end up being able to avail themselves of services that do not require human intervention without being charged for them. So far, this has always worked itself out, if only because hardware support contracts are also covered by the facility; projects living in the grey area usually find that hardware repairs alone make it more economical to come into the facility all the way.

Security vs. Ease of Use

The ITI case study above shows a second common point of conflict; users want to be able to do anything they want to without trouble, but they also want to be safe from malicious others. It is left to the system administrators to provide the security. There is obviously a large technical component to this, but there is also a major political component.

Rules that are technically uncomplicated, like rules mandating that passwords must be changed regularly, or that users cannot share accounts, or cannot have root access, turn out to be emotionally complex. (A user once explained at length in a public meeting that he was too eminent a professor to be required to change from the password he had always used – which had just been broken in the first pass of an automated password tester.)

Security concerns are relatively easy to get management support on; security violations are highly visible in the media, and the technical issues surrounding passwords are easily understood. (At a commercial site, the argument that competitors could exploit security holes to gain access to internal information is extremely effective.) Password changing policies can be approved at a high level, and then implemented impartially in software. Shared accounts can be replaced with groups, usually with minimal resistance.

Root passwords, however, remain a point of contention. Some people actually need them; some people sincerely but incorrectly believe that they need them; and some people want them just as a sort of merit badge, to indicate that they are powerful and competent. Some sites have had success in discouraging people in the latter two classes by giving out root access to machines conditionally; one favorite is a site which requires all people with root access to wear beepers so that they can be summoned to fix the machines when they break.

Some relevant questions:

- What rules are there about choosing and changing passwords and how are they enforced?
- Can multiple users share a single account?
- What does it take to make a machine trusted?
- Can users have **.rhosts** files?

Resource Utilization

It is a recognized law of computing that usage will increase to consume all the available resources; what appears one day to be endless amounts of free disk space turns out to be barely enough on the next. Furthermore, there are cases where resources can be temporarily monopolized, even when they are generally in ample supply. Printers are usually the victims of this syndrome; there's plenty of printing capability, until the day someone prints out accept/reject letters for an entire conference from an automated script, and puts over 200 jobs in one print queue. One such occurrence is enough to produce large numbers of users who want Something Done.

Disk space, printer pages, and CPU cycles are the three most commonly abused computing resources. There are systems for accounting for all of them; these systems differ widely from machine to another, but are more or less uniformly unsatisfying. Most of them simply report the usage, and let you try to figure out what to do about it. Even those that do apply restrictions need to be told which restrictions to apply. Any way you look at it, it turns out to be almost a pure policy decision.

For disk space and printer pages there are two common methods: assign an allocation and cut people off when they go above it, or charge per-page or per-kilobyte in either real or imaginary money. Methods that impose quotas may be impractical, since users with a critical need may run over the quota when nobody is available to restore service to them. It is also tricky to determine where quotas should be set. Quotas need to be high enough so that users do not normally exceed them; on the other hand, they should be low enough so that if people do reach their quotas they do not exceed the available resources. We have never actually seen a system that reliably met both these goals. Instead, quotas are usually positioned where 90 percent of the users fall into them, and resources are allocated so that problems are rare in practice, disregarding the possible results of all users using up their quotas at the same time. Money-based methods, even if they are based on imaginary money, tend to bring out the worst in users. Many become paranoid about getting charged correctly. Since accurate charging is difficult, system administrators may find themselves spending large amounts of time fixing accounting systems which do not really reflect costs. Users also spend a great deal of time and energy questioning the basic accounting structure in hopes of changing it to their benefit.

Informal systems can be quite effective. For instance, OSU-CIS controlled disk space usage effectively for some years by simply publicizing the usage statistics for the top 10 users on any partition that got too full. As long as the largest users on a partition are not also the most powerful, peer pressure is very effective. (The system adopted after that became impractical is detailed in [Zwicky].) Similarly, if you track pages printed, you can deal individually with excessive users.

Some relevant questions:

- How much disk space do users get, and what happens when they overflow it?
- How many pages, at what time of day, on what printer, constitutes fair printer usage?
- How many pages, at what time of day, on what What can you do on someone else's workstation?
- How many pages, at what time of day, on what Who has priority use on public workstations?
- How many pages, at what time of day, on what On a multi-user system, how much of the machine's capacity can you use for what?

Accounts

At first glance, there seem to be relatively few issues about accounts, aside from the security issues discussed above. However, in a multiple-machine environment, there are considerable difficulties in deciding who gets accounts on what machines, as well as the technical problems in reconciling accounts between machines that interact with each other. Technical solutions are a dime a dozen, and come in three forms: network user database services like Sun's Network Information Service (NIS, formerly called YP) or Project Athena's Hesiod; services that provide unique and consistent user ids for a site, which are then used as administrators wish on individual machines; and systems that reconcile password files between machines as users are added (for instance, the one described below).

Some relevant questions:

- Who gets accounts on which machines?
- When do accounts expire?
- What do you have to do to get an account?
- What are accounts named?
- Can a single user get more than one account on the same machine?
- Can multiple users share a single account?

Case Study: ITI

In the past, unofficial policy was to grant user accounts only on the systems needed by the individual user. This kept down the total number of accounts, and made dealing with loosely connected system easier.

As technology progressed, this became more and more of a problem. Cross-mounting NFS systems between hosts with disjoint **passwd** files was a nightmare. Having a user home cross-mounted between systems was difficult due to different setups on different systems.

Over the course of time, a user's needs would change. Accounts once required on one system became inactive, while new accounts were required elsewhere.

We also make extensive use of PC-NFS. The initial installation dedicated a Sun file server to PC-NFS usage, while requiring users to have other accounts on other systems. Disk space crunch quickly made this infeasible, and PC-NFS-mounted directories became intermixed with user home directories. As our user community and our use of PC-NFS became more sophisticated, this became a bottleneck. It also led to such bizarre circumstances as users **ftp**ing files from their home directories to their PC-NFS directories when both were in the same partition.

In addition, each of our central systems is quite different. Vendor and resource constraints constrained us in trying to make them identical; expensive 3rd party software that only ran on one system made it inevitable systems would be different.

The Solution

Briefly, we decided to adopt a rule of "one user, one uid, one home directory". To avoid problems of disjoint access to systems, we decided to change the policy on systems so that all users had access to all central UNIX systems (MIS systems are an exception). This had to be done without use of yellow pages (highly insecure, and not available on all systems) or Hesiod (some systems could not easily be retrofitted). In addition, to defeat previous break-in attempts we were running custom **login** programs with shadow passwords. We were forced to continue with flat files.

Mass implementation would be a nightmare; we didn't even attempt it. Instead we went to a sliding implementation.

All new users were immediately added to a central system. A variant to the new user script was written expressly for the purpose of duplicating a user entry from one system to another. The new user script was run to create the user, then the duplicator run on all other systems. This gave a common home, login name, uid/gid, and common initial password on all systems.

Reconciling the old users was (is) a stepwise process. The machines which were the primaries are gradually being removed from service. As each user is moved to the new systems, his account is cloned. If the uid and login id were unique, they are carried over. If not, new ones are assigned. However formed, the new account is then distributed to all systems. When the old system is decomissioned conflicts with old uids and login names become irrelevant.

Results And Retrospective

The change of policy was justified to management by claiming it would simultaneously decrease administrative cost while increasing user access. This process is still continuing as of this writing; it is expected to be complete by presentation of this paper. The preliminary results are bearing out our estimate.

Requests for accounts on other systems have dropped to almost zero, and will vanish when implementation is complete. This has not only reduced our unplanned administrative tasks, but has also eliminated the problem of duplicated disk space, resulting in more available disk without purchasing additional spindles.

Reconciling system setups was daunting but doable; we're quite proud of the design, implementation, and result of this reconciliation. Previously giving a user an account on a new system immediately led to a flurry of phone calls on what was different where; these have been greatly reduced. At some small per-user cost in loading initial accounts, we have eliminated a great deal of ongoing support. The time and effort expended in designing system-sensitive user initialization files is quickly being paid back.

Without the change in policy, these savings would not have been realized.

Case Study: OSU-CIS

Originally, OSU-CIS maintained a single password file for all workstations that would run both NIS and NFS, ensuring that each user had one account and one home directory. Machines that did not run NIS each had individual password files; user numbers were distinguished by giving each password file a unique range of IDs, and giving an account the first unused ID in the range for the machine it was first installed on. Accounts on the individual machines were given to faculty on request; students had to get a faculty member's signature to get accounts. The machines that had individual password files were primarily the CPU-intense machines (a selection of Pyramids, a BBN Butterfly, and an Encore Multimax). In fact, the Pyramids all used the same password file, distributed from a central machine via **rcp** by **cron**. The Sun servers were not YP clients, and had password files with only staff members in them. To complicate matters, while some undergraduates had permanent accounts, most were given accounts only when they were taking classes; approximately 1,500 of these temporary accounts were created at the beginning of every quarter, and deleted at the end of the quarter.

In order to manage this, OSU-CIS developed two account installation programs (both primarily originally written by Chris Lott). One of them, for regular accounts, allowed you to enter the information about a single user; it then polled each machine which had a password file to determine whether the user already has an account, and if so, tells you the user name and number. You were free to override this, especially since the program might find multiple accounts (usually because of multiple users with similar names, but sometimes because somebody made a mistake). The other one read a tape, produced by the university's registration system, and created a single account for each student on it. The registration tape contained university ID numbers, which allowed that program to be completely certain which were duplicate entries for the same student, and which were entries for different students with the same name. Since this information was not available for existing users, there was no attempt to avoid giving the same student both a regular and a temporary account.

This system, while workable, was inconvenient: limitations on root privileges meant that system administrators tended to deal with user files from the file servers, where the users did not have accounts, so that all the files were shown by numeric ID; mail could not be delivered to students on the central department machines, since those were the CPU-intense machines that the students didn't have accounts on; and password files tended to slowly diverge from each other, as administrators made "temporary" changes. Furthermore, maintaining the password files on the servers became burdensome as the number of servers increased from 1 to 14, and the number of people who needed access went from 8 to approximately 30. On the other hand, there was no interest in changing the fundamental policies about access; giving the world at large access to either the CPU-intense machines or the servers was obviously undesirable. (The per-quarter accounts were a temporary expedient, due to be replaced by a user database allowing undergraduates to have accounts for the duration of their time as CIS majors at OSU.)

Client NIS was enabled on the Sun servers, as well as the clients, but instead of simply pulling in all accounts, two separate lines were added. One pulled in all the accounts for systems staff members, using a netgroup. The other pulled in all remaining accounts, overriding the passwords and the shells. A modified version of **su**, created by Paul Placeway, allowed systems staff to **su** to users without forking the user's shell. Thus, the systems staff could not only see real names on files, they could also run as users on machines that the users could not log in on.

The password files were reconciled with a **perl** program, written by J. Greely, which took each password file in turn, and added lines for the users that were present in the other files but absent in it, with a dummy password and shell, ignoring system accounts. It ran once a night, from **cron**.

Software support

As systems are used, they accumulate more and more software. This has to be installed, upgraded to new version as they become available, ported to new machines as they become available, fixed when bugs are noticed, and explained to users. If software is allowed to accumulate at the whim of users, the tasks involved in supporting it rapidly take over.

Some relevant questions:

- Which programs can you expect the staff to fix for you, and how soon?
- When can you expect to get help, and from whom?

Case Study: SRI

Over the years, SRI's machines had gathered an immense amount of software in /usr/local; we were providing support for any program anybody had ever asked for or purchased. As we moved from VAXes and Sun-3s to SparcStations, we were being asked to port all of this software and continue its support. Some of these programs had no locatable source code; others would not compile; some we objected to on basically aesthetic grounds; and others were simply the third or fourth program to do the same thing. We rebelled, and refused to invest our time in porting four SunView clocks to SparcStations. We then found ourselves embroiled in a political argument.

We developed an 8 page list of software, which we are in the process of publishing to the division. It details exactly what we are willing to support in formal terms, and carries the approval of three levels of management. The list itself is bound to be controversial, but it will get all the arguments over at once. It will minimize users trickling into our office for months, claiming that their lives are incomplete without a really good digital clock for SunView.

Our list currently divides software into 7 categories:

Fully supported: Fully supported tools are considered necessary for day to day life. If they become unavailable, restoring them is first priority. With the noted exceptions, they are available under all versions of the operating system, on all hardware platforms. They are upgraded to new versions regularly, and they are supported by multiple people on systems staff.

Partially supported: These are considered useful, but not essential. If they become unavailable, some priority is given to restoring them. We attempt to make them available on all versions of the operating system and on all hardware platforms. They are upgraded to new versions as time permits, and are supported by at least one person on systems staff.

Available but unsupported: These tools have been installed on some machines. They may not be available on all operating systems or hardware platforms. If they stop working, they may never be fixed. They are unlikely to be upgraded to new versions. Support for them may be unavailable from systems staff.

Under evaluation: A small number of licenses are available for evaluation purposes, or as part of beta-test program. These programs are supported by at least one person on systems staff, but may disappear without warning. They should under no circumstances be used for important or long-term work.

Supported in future: These packages are not yet available, but we are in the process of purchasing and/or installing them.

Supported during transition: These packages are supported because they are still in use, but have been replaced. Users who are already using them are encouraged to move to a fully supported option, and new users should choose a fully supported option. However, those users still relying on transition programs will receive full support as far as is possible.

Completely unsupported: We do not believe that these packages are currently available on our systems. They will not be made available in the future, and any copies that may have escaped our notice are not supported. This category includes programs that we have previously supported, but which are no longer available, and programs which have been evaluated and rejected.

The list is divided up into rough categories (window systems, programming languages and tools, text editors, and so on). Most categories simply include all the programs in the category, sorted by support levels. In some cases, we found it useful to add extra information about what we do and don't support. For instance, we have discovered that people assume that we will lovingly preserve any changes they make to the disks on the supposedly dataless workstations on their desks. Our opinion on the subject is not really repeatable in polite company, so we added a paragraph explaining which changes to a workstation we would and would not preserve.

We also discovered that there were bitmapped backgrounds installed in system space that were not repeatable in polite company either. Since all opinions on the subject of nude and semi-nude backgrounds can be classified as fascist, sexist, or both, we made a blanket decision that we would not install or provide support for images that didn't come with operating system or window system releases, and added that to the support list.

Ideally, this support list should be accompanied by a policy that states who gets to control the list, a question we have so far managed to finesse. The list reflects primarily the opinions of the people who were willing to spend the time compiling and editing it. The process was considerably simplified by having management who understand that it is advantageous to limit the number of programs supported, and to move to new technology as it becomes available. This makes them unsympathetic to users who claim that we need to port the Rand "e" editor to SparcStations "for backwards compatibility".

In the case of programs that must be purchased, there is an unofficial policy that multiple choices will be evaluated by the staff and the users, and a final decision will be made by a group of the primary users for the program, and approved by the people who spend the money. After the public evaluation period, people who object to the choice can simply be informed that they should have spoken up when we asked them to, and that it is now too late. This procedure has been used in the last several major software purchases, and has been quite successful. Our major problem was restraining enthusiastic users who wanted to buy the first program that they tested.

Case Study: ITI

In moving from a loosely coupled to a tightly integrated environment, one immediate problem was differences in utilities from system to system. Our heavily populated VAXes were loaded with things from users, from USENET, and from unknown sources. In order to make users mobile between the systems, we had to somehow deal with these differences.

Licensed software was not a difficult issue. This usually came in object form only, with restrictions that it could only be used on a given system.[1] Users who wished to have some licensed utility on another system were asked to justify the cost of obtaining it for the other system; on learning the cost of same the user usually dropped the request. Other custom but non-sharable items like databases were distributed so as to be closest to their user communities.

Most difficult was the wealth of software that had shown up in /usr/local/bin over the years. This actually became another case of turning a problem into a policy for preventing problems. Previous administrators had been lax in such areas as documenting and archiving these utilities. They had also been fairly firm about not letting users put things into /usr/local/bin. Starting with the installation of a new central system, we established a policy that all programs to be installed must include source. This ensured that it was at least minimally possible to provide a program in other environments.

Programs were broken into 4 categories: vendor-supported (i.e., came with the system), ITI-supported (such as MIS systems, etc), ITI-installed, and user-installed. The last two categories are almost identical, the only difference being in whether the program came because the systems staff thought it was useful or if it came from a user. Neither of the last two categories is really supported, although for ITI-installed the systems staff agrees to at least look at problems and consider fixes. User-installed programs are the responsibility of the user donating the program. If the user leaves the program either becomes orphaned, gets adopted by another user, or (if sufficiently popular) gets adopted by the systems staff.

This last policy has had an interesting effect on programs from users. Previously we had a regular series of requests that amounted to "Gee, I found the neat program. Would you install it?" Now that we say "Yes, but we'll refer questions and problems to you" the response is often "Never mind."

[1]We actually have very little software that has restrictive licenses.

Changing Technology

As time goes by, new computers, operating systems, and programs become available. Usually, the new technology fixes things that were broken before; without exception, it breaks things that worked fine before. Users are usually split between the people who want the newest thing, today, and the systems staff can figure out how to work around the bugs, and the people who never want to change anything. The staff has to hold out until technology becomes reasonably usable, and then has to pry the remaining users off the old technology when it becomes unusable.

Case Study: SRI

SRI is in the process of moving from being based on SunOS 3.5 running on Sun 3s to being based on SunOS 4.1 running on Sparc machines. The process has actually been simplified by changing hardware and software at the same time; the users find it logical that the software should be different on different hardware platforms, for one thing.

Initially, we converted the staff to Sparc, starting originally in SunOS 4.0 Beta. We declined to move users to the new OS until 4.0.3 was released, at which point we moved a few servers worth of Sun 3 clients that either wanted the new operating system, or were purely administrative and did not care which operating system they were running under. We introduced SparcStations as 4.1 Beta came out; users were told that they could have SparcStations running the Beta software, or no SparcStations at all, and quite a few took the deal. When 4.1 was released, we began the move in earnest.

We purchased a Sparc server, and Sparc upgrades for two of our eleven servers. We brought up the new Sparc server, and freed up one of the existing servers by moving clients to other servers, or changing them to dataless SparcStations and moving the relevant home directories to the new server. We then upgraded this server, and the clients and home directories from 3 of the remaining old servers onto it. Two of those servers were decomissioned completely, and their disks re-used on the remaining one. We took advantage of the complete change to make the hardware and software layouts on the servers more consistent as well, which involved re-using most of the racks as well. (The CPUs and the remaining odd-sized racks will be used to upgrade remote sites running on older Sun 3 hardware.) When the third new server was brought up, we moved the clients and home directories from most of the remaining servers onto it, and decomissioned them. Of the remaining machines, one is a staff server, one is a dedicated database server, and one holds the remaining programmers who have projects that cannot be moved to 4.1. Because conditions have changed since we started, the original server needs to be re-configured before we can move the three last clients off the last machine scheduled to be decomissioned, but the move is otherwise complete.

The results have been quite satisfactory. Since each machine ran in parallel with the machines it was replacing for a few days, we were able to go back and fix things that we had failed to move correctly the first time. We were able to introduce some minor changes that increased consistency and security as part of the global change. The users actually have found the change smooth enough so that they occasionally forget it happened, and call us up to ask why they can't log into machines that no longer exist. We did discover some odd side effects of decomissioning central file servers while leaving most of the systems running; mysterious performance problems cropped up, which were eventually traced to machines that were desperately trying to arp for servers that had ceased to exist weeks before. These problems had to be traced by watching the network, since the machines in question had all been reconfigured for the new configuration, but not rebooted.

Enforcing Commonality

The single biggest headache in administering a network of systems is trying to remember the differences from system to system. The obvious solution is to reduce those differences. While this cannot be completely done, enforcement of several simple policies can greatly improve consistency.

Case Study: ITI

As mentioned above, we have established conformance of logins and aliases across networked systems at a given site. In the past it was practice to divide the user community across the various systems to maximize load balancing. This resulted in a nightmare of administrative activity to keep everything ''straight'' on the various platforms. By mandating that all users will have ids on all systems, we have reduced this problem somewhat.

We soon expect to be automated to the point of having master user id/password and aliases files that get distributed to the other systems when updated. A new adduser script has been written not only to generate the ids, initial password, home directories and the other normal functions of such a script, but also to distribute the created entries to the other connected systems.

With common accounts, the next step was to force common NFS layouts. We adopted the /home style for user accounts, such as seen in Sun 4.0.0 and other more recent UNIXes. Each partition on a given system is named after the system and number 1 through *N*. In all cases, an entire partition is given over to a home area. A standard /home directory is present on all systems, with the mount points being

/home/system*N*. This ensures that all homes are identical on all systems. Enforcement is almost trivial, as it is simpler for systems to comply with the policy than to use some other method. This also has the benefit that no matter what system one views the network from, the configuration is identical.

With common IDs and home, there must be a standardized method for delivering mail. Each each user has a home system, defined at this site as the system upon which the user receives electronic mail. While this is usually where the user performs the day to day activities, we do not require this. This is managed by a master alias file which is distributed to all systems and then automaticly localized as need on the individual systems. These common aliases allow ease of managing mail delivery. It also has the curious benefit of allowing an administrator to quickly find the home system for a given user by looking at the alias file.

These policies have proven quite useful across a variety of system types. Our current systems include two DEC VAX 11/785 systems running BSD 4.3; one Encore (nee Gould) PowerNode 6040; a SUN 3/160 file server, a DECSystem 5810 running Ultrix and several PC-based and other small UNIXes. In spite of our best efforts there are system differences, but standardizing disk configurations and user ids has greatly reduced the administration burden. While this has not made our site any more user-friendly, it has made it less user-hostile.

Selling The Policies

How does one go about establishing policies such as those discussed above? Most of the time it is a matter of simply stating it as policy and the great bulk of the users simply follow along. In many cases the users simply don't care or are willing to put up with minor inconveniences (especially temporary ones) if they are assured of better (faster, more understandable, less surprising) systems as a result.

Does management care? If such policies are presented as improvements in user or administrator productivity, management usually eagerly approves. But be prepared to back up the proposed policies with facts, don't exaggerate the benefits: The policies discussed here will not make a system administrator 1,000 percent more productive (though it often seems so to us once things are in place). State reasonable numbers that can be expected. Management loves to hear of 10 and 20 percent productivity gains, but is usually skeptical of 50 and 100 percent.

Be prepared to show that such improvements did occur. Not only will positive and truthful results increase your credibility, and thereby allow management to give you obscene raises in salary, but they really will make your life as a system administrator much more comfortable.

References

Farmer, Daniel and Eugene H. Spafford, ''The COPS Security Checker System'' *Proceedings of the Summer USENIX Conference*, pp. 165-170.

Harrison, Helen E. and Tim Seaver, ''Enhancements to 4.3BSD Network Commands'' *Proceedings of the Workshop on Large Installation Systems Administration III*, pp. 49-52.

Hovell, Bud, ''System Administration Policies'' *UNIX REVIEW*, March 1990, pp 28-39

Zwicky, Elizabeth, ''Disk Space Management Without Quotas'' *Proceedings of the Summer USENIX Conference*, pp. 41-44.

Elizabeth Zwicky is a system administrator for the Information, Telecommunications, and Automation Division of SRI International in Menlo Park, California. She is working on compiling a perfect record as the speaker with the smallest number of slides at every LISA conference. Reach her at SRI International; Information, Telecommunication, and Automation Division; 333 Ravenswood Ave; Menlo Park, CA 94025 or electronically at zwicky@itstd.sri.com.

Steve Simmons is a graduate of the University of Michigan, and has done UNIX-based development at Bell Northern Research, Schlumberger Technologies, and ADP Network Services. He is currently the UNIX systems manager at the Industrial Technology Institute and a consultant. His publications include music, humor, essays, and software. He has published no intentional fiction. Reach him at Industrial Technology Institute; P. O. Box 1485; Ann Arbor, MI 48106 or electronically at scs@iti.org.

Ron Dalton is a graduate of Ohio State University, and has a long career as a systems and software development manager at ITT and Schlumberger. He is currently a systems and MIS manager at the Industrial Technology Institute. Reach him at Industrial Technology Institute; P. O. Box 1485; Ann Arbor, MI 48106 or electronically at red@iti.org.

LISA IV
October 17-19, 1990
Colo. Springs, Colorado

Timothy Howes – University of Michigan

Integrating X.500 Directory Service into a Large Campus Computing Environment

ABSTRACT

X.500 is a proposed international standard for providing Directory Service in an OSI environment. It is intended to encompass nameservice, white and yellow pages service, directory support for electronic mail, and a host of other applications. The problem we face in providing X.500 service to the University of Michigan is two-fold. First, we must define what services should be provided and how those services should be integrated into the existing U-M computing environment. Second, we face the problem of scale. Because of the very large computing community at U-M, we face problems not encountered at smaller sites. These problems have led us to think carefully about the organization of our data and servers, and to make several changes to the prototype X.500 software we are using. This paper describes these changes and services, and our solutions to some problems encountered in implementing them.

Introduction

X.500 is a proposed international standard for providing directory service in an open systems computing environment. The standard is described by two parallel documents, the CCITT X.500 series of recommendations[1] and ISO Draft Standard 9594.[2] Except for minor cosmetic differences, these two documents are equivalent. Directory service promises to play an important role in the OSI computing environment of the future, providing users access to a world-wide wealth of information. By moving to X.500 now we hope to ease the transition to OSI and provide service to our users they would not otherwise have access to.

The University of Michigan has a need for Directory Service in several areas including electronic mail, online white pages service, campus-wide login, uid, and namespaces, elimination of duplicate campus-wide databases, etc. In addition to defining what services X.500 can and should provide to the campus and how these services should be integrated into the existing computing environment at U-M, we face the problem of scale. U-M is comprised of approximately 55,000 faculty, staff, and students who need to be represented in The Directory. This requirement has led us to think carefully about the organization of our data and servers and to make several changes to the prototype X.500 software we are using. This paper describes our efforts to provide a campus-wide X.500 service fully integrated with the existing computing environment and is organized as follows. Section 2 gives a quick overview of X.500 and can be skipped by those familiar with the standard. Section 3 gives some background on the U-M computing environment and the past state of directory service at U-M. Section 4 describes the services we would like to provide via X.500 and how those services are being integrated into our computing environment. Section 5 describes the organization of our data and servers. Section 6 addresses some deficiencies in the X.500 prototype software we are using and describes some changes we have made to the software. Section 7 talks about possible future work, including more software modifications and services to be provided. Finally, section 8 gives a summary of what we have accomplished so far.

What is X.500?

Basically, X.500 is an attribute-based nameserver. It uses a hierarchical, tree-structured namespace to provide facilities for naming, searching for, and retrieving and storing information about *objects*. An X.500 object is one that appears somewhere in the X.500 namespace (either as a leaf or an interior node of the tree), and is composed of a set of *attributes*. Each attribute has a *type* and one or more *values* associated with it. Some attributes have only one value, others have many. One of an object's attributes is designated special and is used to form the object's *Relative Distinguished Name,* or RDN. An object's *Distinguished Name* (DN) is the concatenation of the object's RDN with the DN of its parent (the object that appears just above it in the tree) and is used to unambiguously refer to the object. A DN denotes exactly one object, and each object has exactly one DN.

Object classes are defined in terms of the attributes a member of the class *must* contain and the attributes it *may* contain. There can also be a parent object class associated with an object class, all of whose attributes are inherited by the child object class. Note that here, "parent" and "child" refer to object class inheritance, not to the relative position of any objects in the X.500 namespace. For example, an object of class **person** must contain the attributes **commonName** and **surname** (both of type **CaseIgnoreString).** It may contain the attributes **description, seeAlso, telephoneNumber,** and **userPassword.** The object class **organizationalPerson** is derived from **person,** inheriting all of its attributes. An **organizationalPerson** may also have a **title** and other attributes used to specify office address, telephone number, and other information.

The X.500 Standard defines two protocols. The first is called the Directory Access Protocol (DAP) and is used for communication between clients (termed Directory User Agents, or DUAs) and servers (termed Directory System Agents, or DSAs). This protocol defines the operations the directory can perform, including **read, list, compare, modify, modifyRDN, add, delete, search, abandon,** and **bind.** Much of the power of X.500 is contained in its search capabilities. A search can be carried out on a single entry, its children, or an entire subtree. Complex search filters can be constructed to compare any number of attributes for equality, inequality, approximate equality (e.g., soundex matching), and arbitrary boolean **(and, or, not)** combinations of the above. For all operations the directory can perform, a number of *service controls* can be given to tell whether aliases should be dereferenced during the operation, to specify a size or time limit for the operation, etc. The Directory also has quite extensive security provisions allowing public key cryptography style authentication, and digital signing or encryption of data. The second protocol is called the Directory System Protocol (DSP) and is used for communication between DSAs for such things as chaining of requests and management of replicated data. The Directory is a distributed application, providing the ability to spread data across many DSAs while providing users with a single homogeneous view of the namespace. The DSP is used to provide this transparency.

The Standard also defines many commonly used attribute types and object classes. The system is extensible and allows users to define their own object classes and attribute types in addition to those defined in the Standard. Also described is a suggested structure for organizing data within the X.500 namespace. The suggested namespace organization places **country** objects at the top level of the tree, followed by **organization** or **locality** objects at the next level. Below this level lie **organizationalUnit** or more **locality** objects, followed by people or more organizational units or localities. For example, the DN of an entry belonging to someone in U-M's Information Technology Division would look something like this:

c=US@o=University of Michigan@ou=Information Technology Division@cn=first last

where **c, o, ou,** and **cn** are abbreviations for the complete attribute names (the Standard allows one valid abbreviation for each attribute type). Also fitting into this hierarchy are **applicationEntities, applicationProcesses, devices,** and more. Augmenting the hierarchical namespace is the capability to add **alias** entries. An alias entry points to another entry in the tree and can be used, for example, to reflect a person's dual role or appointment in two different organizational units without having to replicate the entire entry.

Directory Service at U-M

The University of Michigan computing environment consists of a few large mainframes (e.g., IBM 3090-600E), about 1500 workstation-class computers (e.g., Sun 3's), and several thousand Macs, PCs, and other small machines. Over 2000 of these machines are connected to the Internet. A great deal of both intra- and off-campus electronic mail is sent to and from the larger machines, particularly one of the mainframes running an OS developed at U-M called MTS. Historically, directory service on campus has been neither centralized nor coordinated, and is limited to rudimentary white pages information. Each medium or large host typically keeps its own database of user information (e.g., /etc/passwd on Unix machines, *USERDIRECTORY on MTS) which is accessible by host- or OS-specific means (e.g., finger, RUN *USERDIRECTORY).

MTS and *USERDIRECTORY used to provide a sort of campus-wide directory by virtue of the fact that "almost everybody" had an MTS account. This directory was useful for both "friendly" electronic mail addressing, and white pages directory service, though until recently none of these services were network accessible. Even now, many users who have moved off of MTS to do their computing elsewhere keep accounts on the mainframe and an entry in *USERDIRECTORY to route their electronic mail to their workstation. With the growing number of workstations on campus and the continued movement away from mainframe computing, using *USERDIRECTORY as a *de facto* campus-wide directory is no longer a satisfactory solution.

Integrating X.500

Electronic Mail Support

We are integrating X.500 Directory Service into the U-M computing environment in several areas. The first and perhaps most visible area is electronic mail. Our goal is to provide a stable campus-wide electronic mail address namespace that will free people from having to know which machine a user has an account on and what their login is in order to send that user e-mail. With this "business card" e-mail service, all a user will need to know is the name of the person to whom they want to send mail, and that the person is somewhere at the University. X.500 will be used to provide a mapping between more friendly user-level names (e.g., *Tim.Howes)* and the sometimes cryptic and frequently changing RFC 822-style *login@hostname* mail addresses (e.g., *tim@terminator.cc.umich.edu).* We also want to provide approximate matching capabilities and to generate more friendly "bounce" messages in the event of a soundex only match to one or more names. (An example bounce message is given in the appendix.) For example, if a user were to send to *Tim.Hawes* and no Tim.Hawes existed, a bounce message should be generated explaining the problem and listing possible matches for the ambiguous name specified, along with some additional information to help choose which of the names is the intended recipient.

To implement the above plan, we have made modifications to *sendmail,* the primary UNIX mail transport agent. For incoming mail, our modifications cause *sendmail* to formulate a query that searches the X.500 database for entries whose **commonName** matches the user portion of the recipient's electronic mail address. Upon finding a match, the **rfc822mail** attribute is returned and treated just like an alias for the original address. In the case of multiple matches (because of a soundex collision), disambiguating **title** attributes are returned for inclusion in the bounce message along with the names that matched. The **title** attribute should contain a description of the person or the job they perform useful in distinguishing that person from other people with similar names. For outgoing mail, just the reverse query is formulated. The X.500 database is searched for entries with an **rfc822mail** attribute matching the address of the sender. If an entry is found, the sending address is rewritten as the common name of the entry. For example, if mail is sent to *Tim.Howes,* the incoming X.500 lookup would return *tim@terminator.cc.umich.edu,* which is the address used by sendmail to actually deliver the mail. If *tim@terminator.cc.umich.edu* sends mail, the sending address would be looked up in The Directory, returning Tim.Howes, which would then be appended with the top-level domain name to form *Tim.Howes@umich.edu,* the sending address the recipient of the message would see.

These modifications have been in place for almost three months, serving about two dozen users on a machine called **terminator.cc.umich.edu** (not at the **umich.edu** level – that is, mail is accepted to and rewritten as *first.last@terminator.cc.umich.edu,* not *first.last@umich.edu).* Soon, we will provide this service for all of U-M's Information Technology Division, and eventually at the **umich.edu** level for the entire campus as described above. Our intention is to provide a subscribable service. Users will be able to subscribe to the "business card" portion of the service, allowing them to receive mail at the **umich.edu** level, and they may also choose to subscribe to the outgoing service, making any mail they send appear to come from a name at the **umich.edu** level. Most of our problems in providing this service to the campus at large are political and administrative rather than technical (e.g., obtaining and entering the data, deciding on a name collision resolution policy).

White Pages Support

A second service long-desired on campus is an online white pages directory through which users can look up the phone numbers and addresses of their friends and colleagues. As mentioned above, this service is currently provided on a per-host basis through a number of different interfaces (finger, whois, run *USERDIRECTORY, etc), accessing a number of separately administered databases. Our goal in this area is to continue to allow people to use the interface they are most comfortable with (as much as possible), yet have all these interfaces access a common X.500 database. To this end, we have acquired one X.500-based white pages front end and developed two more.

First, the *whois* service provided by the NIC is replaced by a program called *fred,* written by Marshall Rose as part of the NYSERNet White Pages Pilot Project, of which U-M is a part (more about this later). The whois database is a centrally administered repository of white pages information, a sort of "Who's Who" of the Internet. Fred is compatible with the whois program, while taking advantage of the distributed nature and more extensive searching capabilities of X.500.

Second, we have developed a replacement for *fingerd,* the UNIX daemon that implements the finger server protocol. Our version of the daemon searches the U-M X.500 namespace for a name matching the request. If only a few matches are found, finger-like information is returned for each entry. If many matches are found, only the names of the entries are returned. The finger protocol specifies that a null user name corresponds to a request for information about users currently logged in to the machine. Since this information is not available in the campus-wide directory, an error message is returned instead. Our finger replacement, called *xfingerd,* is currently running on a machine named **umich.edu.** Users interested in fingering someone at U-M do so with a request like *finger first.last@umich.edu* or just *last@umich.edu.*

Third, we have developed an X.500 client called *dixie* to help ease the transition to X.500 for MTS users. Dixie accepts and displays information and commands in a form very similar to that used by *USERDIRECTORY on MTS. Dixie comes in two flavors: a simple command line version, and an X-windows based version. Both versions of dixie will run on any BSD UNIX machine. For efficiency, dixie does not talk to an X.500 server directly. Instead, it talks to an intermediate server that keeps a connection to the directory up all the time, and is responsible for interpreting the full-blown X.500 protocol. There are three reasons for taking this approach. First, the overhead of setting up an ISO transport connection is not small, and there would be a second or two wait for each client if each was forced to make its own connection to the directory. Second, the dixie client can be greatly simplified if it does not need to deal with the full X.500 DAP protocol, including ASN.1 encoding and decoding. With our approach, all this overhead is borne by the intermediate server, which uses a simple text-based protocol to communicate with clients. Third, because the dixie client-to-intermediate-server protocol is TCP/UDP based, a potentially wider variety of machines can have access to the directory without having to speak ISO protocols. For example, one of our future goals is to provide a dixie-style client on Macintosh and PC platforms (we envision a Directory desk accessory for the Mac).

Data and Server Organization

Since U-M first started using X.500 as part of the NYSERNet White Pages Pilot Project in August of 1989, we have been the largest single X.500 site in the world, holding data for approximately 55,000 entries, representing every student, staff, and faculty member at U-M. This presented us with some interesting challenges and an opportunity to exercise more fully X.500's distributed data management capabilities. In particular, it forced us to think carefully about how many DSAs we should run and how they should be configured. We were further handicapped by the limitations of *Quipu,* the prototype software available to us as part of the White Pages Project. Quipu [3] is a freely available implementation of the X.500 protocols developed by Steve Kille et al. at the University College of London, and distributed as part of Marshall Rose's ISO Development Environment (ISODE).[4]

U-M employs about 20,000 faculty and staff and enrolls approximately 35,000 undergraduate and graduate students. Obtaining data for all these people, converting it to a form acceptable to Quipu, and fitting the data into the X.500 namespace was a formidable task. We wanted a namespace that would be intuitive to users (i.e., one that would make it easy for them to find what they were looking for), efficient to search, easy to maintain, and simple to make changes or extensions to. In order to facilitate user browsing through the directory, we chose to organize the data along externally visible University organizational lines, rather than along internal budgetary lines. Search efficiency is maintained by having only two levels in our subtree and by careful allocation of the data to DSAs (see below).

Still, our solution is far from perfect. Our data comes from U-M's Data Systems Center (the administrative computing arm of the University), and contains roughly the same information found in the staff phone directory plus similar student data. The software we have developed for converting this data to Quipu format does not deal well with dual appointments. The result is that our data contains many duplicate entries for people with dual appointments instead of making a single entry (perhaps in the major appointment unit) and then aliases to this entry in the other units. Also, in practice a fair amount of manual intervention is necessary to remove offensive characters in the data. Recent changes to the conversion software have greatly reduced this necessity, however. Finally, there are many small sub-units within the University, and it is not always clear where these units should appear in the namespace, requiring some manual intervention and arbitrary decisions to be made during data generation.

Quipu uses very crude techniques for managing data within the DSA. It keeps all the data it holds in memory, organized as a tree of linked lists of sibling entries. This technique puts a practical limit on the number of entries a single DSA can hold, depending upon the type and configuration of machine the DSA is running on. For our volume of data, we decided we needed to run four or five DSAs, each holding roughly the same amount of data. Things were complicated by the fact that the chunk of data belonging to U-M's College of Literature, Science, and the Arts is almost 20,000 entries, more than could be handled easily by any mortal machine. Splitting this data between two DSAs would have been possible, but such a split could not have been kept transparent to users. Furthermore, we wanted to organize the data so as to minimize network traffic between DSAs during searches of the entire U-M subtree.

The first problem was solved by porting ISODE and Quipu to AIX/370 running on an IBM 3090-600E under VM (no mortal machine, to be sure!). The 3090 had no problem handling the approximately 40 Megabyte Quipu process. The second problem (that of reducing network traffic) was more complicated and required an understanding of the way X.500 searches, referrals, and chaining work. Conceptually, when a DSA searches a subtree it begins at the root and applies the search filter to the root node of the subtree and then recursively to that node's children. If the DSA encounters an entry whose children it does not hold, it *chains* the search operation to the DSA holding the entries. Network traffic and thus search time can be reduced significantly by minimizing the number of chaining operations that must be done. Obviously, from this perspective, the ideal arrangement is to have all the data contained in one DSA. But this solution is impractical as explained above, and defeats the administrative and fault tolerant advantages of distributing the data.

Instead, we tried organizing our data such that only a small number of chaining requests were necessary and were only performed for large units of data. To this end, we chose a single DSA (Woolly Monkey) to be the main contact point for DUAs. Most search requests start at this DSA, which holds all of the University's smaller units of data (totaling approximately 13,000 entries). Other larger units we distributed among "second-line" DSAs. This method is efficient because with our data organization the DSA chains requests on an organizational unit basis. Because Woolly Monkey holds the majority of organizational units, it does not need to chain requests for those units. When it does need to chain to another DSA, the units are very large (e.g., more than 5,000 entries), making the network overhead a smaller percentage of the total search time. The result is greatly reduced network traffic and subtree search time.

Software Modifications

As stated above, Quipu is not the ideal X.500 implementation for our needs. However, it does work, implement the Standard faithfully, provide some valuable extensions to the Standard (e.g., access control, replication, wild-card search capabilities), and it's free! For these reasons, we wanted to see if it was possible to modify Quipu to make it more suitable for our needs. In the process, we have also made numerous bug fixes. The nature of all of our changes is not to make Quipu deviate from the X.500 Standard, but rather to make it run faster on large data sets.

The first problem we encountered had to do with the fact that Quipu reads all of its data into memory and keeps it there. In doing so, it checks to ensure that each set of siblings it reads contain no entries with duplicate RDNs, which would be a violation of the X.500 naming standard. This process involves a search of all sibling entries for each entry loaded. With N entries, the loading process takes $O(N^2)$ time. Worse yet, the time can increase by several orders of magnitude on a machine with insufficient real memory, since a large Quipu process will begin to thrash as the size of the entries to be searched on each load becomes greater than available real memory. Our quick solution to this problem was to have Quipu build a separate, more efficient structure used only during loading to search for duplicates. Because our data happened to be generated in alphabetical order, we chose to implement an AVL (height-balanced) tree to perform this function. This choice drops the load time to $O(N \log N)$. The AVL tree can be much smaller than the entire linked list of entries because it need only contain the RDNs of the entries, not the rest of the attributes. Furthermore, we allocate memory for the AVL tree in large contiguous chunks to minimize paging activity. These modifications have been incorporated into the official Quipu release and are in use by other large sites as well. One site found that with our changes their load time was reduced from 12 hours to just over 5 minutes, quite an improvement!

The second problem we encountered was in making modifications to the data. Although Quipu keeps its data in real memory, it reads it from disk files when it starts up. One disk file (called an Entry Data Block, or EDB file) exists for each set of sibling entries in the tree. When an entry in Quipu's in-core database is modified, the entire EDB file containing the entry is written synchronously out to disk as well. This method works fine for small sites, but at U-M we have one EDB file with almost 20,000 entries, and several with over 5,000 entries. Writing this much data to a disk file can take several seconds or even minutes on a heavily loaded machine. The result was very poor performance when making modifications to the database. Our solution to this problem was to change Quipu so that it keeps its data on disk in *dbm* files (actually *gdbm* files) rather than the plain text files it currently uses. (*Dbm* is a set of UNIX library routines that implement a simple single unique-key database.) This allows Quipu to randomly access and update any single entry in the file without having to write all the entries. The performance increase, as expected, is quite dramatic. Whereas before, modifying an entry could take minutes to complete (depending on the size of the EDB file the entry lived in), it now happens almost instantaneously and the time does not vary with the size of the EDB file. These changes are in the process of being incorporated into the official Quipu release.

Future Plans

Our future plans for X.500 on campus involve two parts: new and expanded X.500 services, and more modifications to Quipu so that it runs more efficiently. Offering new and expanded X.500 services to the campus is our first goal, having made

Quipu perform tolerably well. Most of our plans in this area involve expanding the prototype services we have already developed and making them available to the campus at large. We want to move our X.500 electronic mail support to the **umich.edu** level, and port our X.500 user interfaces to a more widely accessible class of machines (e.g., the Mac). We have also been experimenting with the idea of providing a network-accessible X.500 server available to the general public.

Eventually, we would like to expand the finger service so that it responds correctly to the null login name query and prints out information about who's logged in where. This involves modifying the login program (and its equivalent on other machines) to authenticate the person logging in and to modify that person's entry in The Directory to reflect their current location. If this service is to be provided, it would likely have to be on a voluntary optional basis because of privacy concerns.

There are still many improvements to be made to Quipu. First, the way data replication is handled is sub-optimal. It is in a master/slave arrangement using a total refresh update strategy where slave DSAs call the master DSA. It would be much more efficient to use a differential refresh strategy, where only those items actually changed are updated in the slave, instead of the entire EDB file. Second, Quipu needs the concept of inheritance in its database. As it stands, if every person at U-M has the same access control list associated with their entry, that information is duplicated in each entry. It would be a tremendous space savings if there was the concept of a ''default'' attribute associated with an entry, the default being taken from the entry's parent, for instance. The authors of Quipu are currently looking into this problem and plan to provide a solution. If they do not, we certainly will, because it would mean a tremendous improvement in our performance. Third, because of the environment in which we are using X.500, we would like to be able to tailor the DSA to be more efficient on some kinds of queries at the expense of others. For example, in the **umich.edu** electronic mail address namespace project, queries on people's names and e-mail addresses are quite common, while other kinds of queries are not. It would be nice to be able to optimize queries on these attributes. We envision a run-time tailor option that one could use to specify which attributes one wanted to optimize. Finally, there is the question of making Quipu disk-based. The authors of Quipu advise against such an undertaking, claiming that Quipu was not designed with this in mind. However, if Quipu is going to continue to be used widely as X.500 grows in popularity, it is clear that its memory-based data scheme will have to go.

```
Date:    Mon, 02 Jul 90 12:24:50 EDT
To:      <tim>
From:    Mail Delivery Subsystem <mailer-daemon>
Subject: Returned mail: Ambiguous user
---------------------

The name you specified is ambiguous.

Possible matches for tim hawes

Tim Howes
        Unix Project Programmer, ITD Research Systems
        PhD Graduate Student, EECS Department

    ----- Unsent message follows -----
Received: from terminator.cc.umich.edu by terminator.cc.umich.edu
        (5.61/1123-1.0) id AA07278; Mon, 2 Jul 90 12:24:50 -0400
Message-Id: <9007021624.AA07278@terminator.cc.umich.edu>
To: Tim.Hawes
Subject: test
Date: Mon, 02 Jul 90 12:24:48 -0400
From: tim

Bounce this message!
```

Appendix – Example bounce message

Conclusion

X.500 Directory Services is still an evolving standard. The standards body recently adopted a resolution to add access control to the Standard, and work is in progress on a replication addendum. Despite its continuing evolution, it is clear that X.500 will provide a valuable service in the OSI computing environment of the future. By integrating X.500 into our present computing environment, we will be in a better position to serve the needs of our user community both now and in the future. We have developed software to integrate X.500 into our electronic mail environment and to provide white pages access to The Directory. Our work to date has been mostly of a prototype nature, and the success of this work has encouraged us to begin to release these services to the campus at large in the near future.

References

1. *The Directory – Overview of Concepts, Models, and Service,* Recommendation X.500 December, 1988.
2. *Information Processing Systems – Open Systems Interconnection – The Directory – Overview of Concepts, Models, and Service,* International Standard 9594-1 December, 1988.
3. Steve Kille, *The ISO Development Environment: Users Manual – Quipu.* 1990.
4. Marshall T. Rose, *The ISO Development Environment: Users Manual.* 1990.

Tim Howes received a B.S.E. in Aerospace Engineering in 1985 and a M.S.E. in Computer Science and Engineering in 1987, both from the University of Michigan. He is currently a Ph.D. candidate in U-M's department of Electrical Engineering and Computer Science, studying load balancing in very large distributed databases. He is also a Systems Research Programmer for U-M's Information Technology Division, where he works on campus X.500 and X.400 protocol support and integration. He is a member of ACM and IEEE. Reach him at 535 West William Street; Ann Arbor, Michigan 48103-4943 or electronically at Tim.Howes@terminator.cc.umich.edu .

LISA IV
October 17-19, 1990
Colo. Springs, Colorado

Helen E. Harrison – SAS Institute, Inc.

A Domain Mail System on Dissimilar Computers: Trials and Tribulations of SMTP

ABSTRACT

SAS Institute's computing facilities include several hundred computers from many different vendors. These computers had communicated through electronic mail in an inconsistent and incomplete manner. A group of system programmers got together and implemented a mail system based on an Internet Domain Name Service and SMTP which encompasses most major computer architectures and provides seamless mail service to the entire company.

Where We Started

Most electronic mail within SAS Institute was based on an in-house mailer developed for the MVS operating system called SMAIL. SMAIL was a simple mailer designed to solve an immediate problem, and was not intended as a long term solution. It is an unconfigurable front end to ftp which does not comply with Internet mail standards (RFC822). As will happen with temporary solutions, SMAIL became the standard in-house mailer, defining the way users viewed electronic communication. SMAIL was ported to other platforms including VM, Prime, VMS and Apollo. Users could send mail to a predefined list of nodes from each SMAIL client. Eventually, the SAS System was ported to several UNIX platforms. As UNIX became a growing market and the number of UNIX machines increased, the lack of communication with the SMAIL system was becoming a problem. As one might expect, there was not much interest among UNIX programmers in implementing an archaic mail system just to send mail to MVS. After all, UNIX has a perfectly functional, extremely configurable mail system which is quite sufficient, thank you. As a first step we attempted to get mail forwarding working to UNIX from the Apollos, our most similar neighbor. The mailer of choice on the Apollos is DPSS, a screen oriented mailer which, until recently, only talked to other Apollos. An Apollo system programmer had written a program to convert an SMAIL message into RFC822 format, and a program to read DPSS mailboxes and forward messages to sendmail. This allowed DPSS users to receive SMAIL and SMAIL users to send mail to UNIX users who had their mail forwarded from the Apollos. A cron job ran every 10 minutes to look for mail to forward, based on a list of users and their .forward's. Unfortunately the mail got sent to postmaster as often as to the intended user. Even when this worked, this only solved half the problem. Mail flowed only in one direction, from the mainframe to UNIX. There had to be a better way.

The UNIX community communicated happily within itself. We even had a uucp connection to the Research Triangle Institute (`rti`) for mail and news. Unfortunately we lacked an error correcting modem and noisy phone lines were causing most uucp sessions to terminate abnormally. Being aware of Telebit's half price deal for registered internet domain sites, I promptly registered the `sas.com` domain, with `rti` acting as our gateway, and ordered a Trailblazer. Sas.com officially existed to the outside world; now it was time to make it work internally.

I convened a meeting of the system administrators responsible for mail or communications on each system to discuss a unified mail system. When all the relevant parties were identified, we had system programmers from UNIX, VM/CMS, MVS, VMS, and Apollo, all in the same room, trying to agree on a solution. This, in itself, was occasionally more challenging than the technical problems we were to encounter.

The Plan

All parties agreed that connectivity was a good idea, but how to accomplish it was unclear. The solution must meet the following requirements:

- Co-exist with current mail mechanisms
- Be flexible enough to allow member systems to join in when ready
- Provide user-friendly addressing
- Be more manageable than current strategies

Computing facilities at SAS Institute include over 900 hosts on 12 networks. The general breakdown of these machines is roughly 500 Apollos, 175 UNIX computers, 20 VMS VAX's, 12 PC's with TCP/IP (and 200 others on Novell where we do not see them), a large Prime machine, a large Data General (AOS/VS) machine, an IBM 3090 hosting MVS and VM (and AIX, but we count it as UNIX). All these hosts are cataloged in a central host table, local.txt, kept on an Apollo. Other hosts copied this table, converting it to the appropriate format. All of these hosts had unique names in a flat name space. In the SMAIL system users are reached as *person@node*. SMAIL is only a transmission program and does not include facilities for forwarding or replying to mail. The preferred SMAIL node for an employee is recorded on the company phone list, which is sent out monthly. Clearly a mail system which encompasses 700 hosts and 1500 users must have some type of centralization. We first considered a central database approach for the `sas.com` domain in which a single node could forward all mail to the correct destination. While this would be a very convenient solution for our users, we dismissed it quickly since we could not maintain it accurately within the current administrative structure. We had to decide how to subdivide the network. We considered dividing machines by department or division. This solution, too, would be convenient and intuitive for our users, but it was unmanageable because some department's computing resources crossed administrative boundaries. We settled on a subdomain scheme which divided our computers along administrative, operating system boundaries. This solution had one outstanding feature: we all agreed on it. We identified 6 domains:

- `vm.sas.com` - VM system
- `mvs.sas.com` - MVS system
- `vms.sas.com` - VMS machines
- `unx.sas.com` - most UNIX machines
- `dev.sas.com` - development machines (Apollos et. al.)
- `pc.sas.com` - varied collection of personal computers

The `pc` domain does not support mail, and so was added for network addressing purposes only. There were also Data General and Prime SMAIL nodes, but they represented too small a user community to justify their own domain so they were put in the `dev` domain. There was concern among the project members that since these new names were longer than SMAIL host names there would be resistance on the part of our users to switching to the new naming scheme. We agreed that for internal use our mailers must support an abbreviated form of the domain names, e.g. `unx` for `unx.sas.com`. In addition, members felt it important to support the current SMAIL names with the closest equivalent behavior.

We decided that it was reasonable to have one machine in each domain which would know about all the users in that domain, and forward mail appropriately. Thus a user would send mail to *user@subdomain*. With this scheme, there would be 5 virtual mailstops encompassing all mail destinations. While the correct destination for a specific user might not be obvious initially, this would be recorded on the phone list where our users were already accustomed to looking.

The Implementation

While the concepts of networked electronic mail, subdomains, and name service may be old news to large UNIX sites, it was very new to some of our mainframe neighbors. Much of our time in early meetings was spent learning new terms and overcoming language barriers. The first problem we ran into was how to implement domain naming given our current network mechanisms. None of the other domains had name service readily available. This meant we had to maintain the new system using the existing host table. We assigned each machine on the network a new entry with its fully qualified name. The task of identifying each of 700 machines by domain was significant in itself. The new entry was added as the primary name so our mailers would pick it up as a return address. The seemingly simple change of adding new aliases caused an unexpected problem. Typically, workstations had friendly names like `warlock` or `tut`. In addition, some machines which were designated as gateways or public machines also had an alias which followed another standardized naming convention as well, derived from its function. Users were not consistent in which name they used since both were equally available. Unfortunately, the TCP/IP package in production on our IBM mainframe only allowed for a single name and one alias. These public access machines now had at least 3. We had to choose which name would not be available and notify users. What appeared to be a simple task took several weeks. It was our first of many lessons in coordination and compromise.

Once we had a naming system in place, we had to find ways to talk to each other. Communicating between Apollo and UNIX was fairly simple, since each had sendmail. VMS had the Wollongong TCP/IP package, which included an SMTP based mail package. The IBM mainframe, however, did not have a production SMTP available. The IBM networking support programmers (a different group than MVS systems support) had begun evaluation of a replacement TCP/IP package from IBM in the fall of 1989. This package included an SMTP server which was put into service in February 1990. VM had been using the RiceMail package locally since early 1989. RiceMail depended on an external SMTP agent to communicate with anyone other than MVS. Since an SMTP agent was not available there had been little motivation to make it a production system. In March 1990 the VM Network MAILER (also known as the Columbia Mailer) was installed as an interface between RiceMail and SMTP. This added services and flexibility in address rewriting that was not available with the other two. MVS had a similar situation. The UCLA/Mail package had been installed for a couple of years. Without an external SMTP agent it could only communicate with VM. These mailers had a variety of different limitations, typically in configurability and address rewriting. Still, all domains had the basic SMTP services necessary to get started.

Getting the gateways of each domain to communicate with each other was straightforward. We had very little trouble, if any, with communications among the SMTP servers on different platforms. As the project progressed it became clear, however, that coordination was critical. One site could not start sending mail to a particular address unless the mailer on the other end was ready to receive it. From the UNIX side, initial stages involved rewriting addresses individually for each gateway in a way that each could accept it. Sendmail is, of course, well suited to this task. Other mailers were somewhat less configurable. We also found that we had the problem of what to do with unknown addresses. An address may be unknown to the current host, but may be valid to a more informed mailer, (e.g., mail destined for Internet sites). A capability that had to be configured into each mailer was a default mail destination for addresses which were not in the local host tables, typically the UNIX gateway. Even this was not a simple task for some mailers.

Next, with basic connectivity established, we had to address the original goal of integrating with existing mail mechanisms. On Apollo and VMS where the SMTP-based mailer was already in production use, the decision was that SMAIL integration was no longer a priority; users should learn the new mail addressing system. On VM, having others with whom to speak SMTP was a significant motivation to put the smtp product into production use and convert users to RiceMail. They quickly mounted an effort to introduce RiceMail and to phase out SMAIL, and also declined SMAIL's integration with the new system. MVS, however, posed quite a different problem. It had by far the largest user base, by almost an order of magnitude. Converting that many people to any new mailer is a large undertaking. In addition UCLA/Mail had only a line oriented interface. A project was already underway to write a screen oriented front end to SMAIL known as PMAIL. As the mail project progressed, emphasis was shifted to make PMAIL a UCLA/Mail interface as well. It is hoped that this more appealing interface will help lure users away from SMAIL. Still, MVS would not be ready to go into production as soon as the rest were, which necessitated an SMAIL connection into the new system. The Apollos already had a way to convert RFC822 mail to SMAIL, and vice versa. The program needed some strengthening, but provided the basic functionality needed. The Apollo sendmail was configured to send mail destined to `*.mvs.sas.com` through this filter to be delivered through SMAIL. On MVS, SMAIL was reconfigured to send mail addressed to any of the new sas domains names via SMAIL for those who already supported it (`vms`, `vm`, `dev`), and send any other "dotted" address, such as `unx.sas.com` or `mcnc.org`, to the Apollo gateway to be forwarded by Apollo's sendmail.

The new mail strategy greatly simplified mail between UNIX machines. The previous approach had been to distribute a /usr/lib/aliases which contained the correct mailstop for each user. As the number of workstations multiplied these alias databases became rapidly out of date. The UNIX computers at the Institute are maintained by several people in different departments. This added additional headaches. Prior to the formation of the mail project, a central accounting machine was designated on which all UNIX users would have an account. All other machines used the same uid/gid for their own users. When mail changes began, these other machines were given a simple sendmail configuration file which did little other than deliver mail to a specific UNIX host if requested, and forward anything else to the `unx` domain gateway which, conveniently, was also the central accounting machine. Except for small enhancements, these subsidiary configurations remained the same throughout the evolution of internal mail. Changes and special addressing rules were handled on the gateway. All users were told to make sure that mail on this central machine was forwarded to the correct place. They were also encouraged to send mail to UNIX users as *person*@`unx` to make sure that mail went through the gateway until .forwards could be put in place everywhere else.

To support the domain naming scheme, and address host table maintenance in general, we started using name service on all hosts which would support it. Since the master host tables were kept on the Apollos we had to be able to generate the named database dynamically. The hosts table were built by taking the local.txt file from the Apollos and converting it into /etc/hosts format. Next we would run an HP-UX utility "hosts_to_named" to generate the named database. The local.txt database is in standard NIC format in which there exists a "services" field. All Institute hosts which support it indicate SMTP service in this field. This information is used to generate MX records for named. Sendmail uses this SMTP information to route mail. Generally, in domains other than `unx`, only the gateway machine supports SMTP. Due to named's flexibility, we are also able to configure a static component of its database for exceptions which were not represented on the host table, such as the designated mail gateway for each domain (only UNIX is running name service – there is no one else to ask about other domains), and aliases or CNAMES for each of the previous SMAIL nodes which we had agreed to support.

The basic strategy had been that mail would be exchanged between domains only though designated gateways with mutually agreeable addressing formats. What happened within a domain was entirely up to its mail administrator. When the UCLA/Mail group was ready to start testing SMTP delivery this presented an interesting problem: two mail stops in one domain. So, we created an exception. An address of `ucla.mvs.sas.com` would be sent to MVS using SMTP. Any other form of MVS address would still be delivered via SMAIL. This was implemented via a special rule in the gateway's sendmail.cf.

Connectivity had been well established between domains, and, although not available (or at least not publicized) to all users, the mail administrators were using it frequently. It was time to add some polish to the system. Security and public image are of particular concern to the Institute. We felt it important that mail leaving the company present a consistent return addresses and not advertise internal hosts. To this end, the sendmail configuration on our UNIX gateway (which still had the external uucp connection to `rti`) was modified to strip the hostname portion of the return address, leaving only the domain. To the outside world we would appear as 5 mail domains. We wanted our users to have the same idea. Wherever possible, addresses were changed to have the preferred form, no matter what the user actually typed – sort of a subtle hint. There was an interesting problem here. If a UNIX user sent mail to `joe@vms.sas.com`, sendmail would have to rewrite this address to `joe@sdcvms.vms.sas.com` before sending it (since this was what VMS recognized as itself). Unfortunately, joe would see the full machine name as his address. This was the opposite of what we were trying to achieve. It turned out that in order for all mailers to accept just a domain name as itself, we would have to add this name as an alias in the host tables. Named on UNIX did not think these aliases were valid so they had to be removed before the database was rebuilt, but this was not difficult. Another problem solved! Now a user in any domain could send mail to another user in another domain as `jane@unx` or `jim@dev.sas.com`. The remaining issue was return addresses. For consistency we wanted the ''From'' address to contain only the domain. This required mailer reconfiguration in each domain, but each could find a way to do it. Now all we had to do was document...

What we learned

The mail project was an instructive exercise in ''inter-denominational'' cooperation. Beyond the accomplishments of the project itself, the experience gained in learning to work together and demonstrating to ourselves that we can work together made the effort worthwhile. Any site with a large, truly heterogeneous network could benefit from such an undertaking. The first mail meeting was held at the beginning of April. By the beginning of July we had a functioning system. It is interesting to note that we were able to implement this system on all hosts from existing software. These packages often needed modification but did not require that we acquire anything new. Here are a few things that we learned that may be useful to anyone who wants to attempt a similar project.

1) Find a sendmail guru; you will need one.
2) Coordination and compromise are critical. Don't worry. It doesn't hurt much. (By the time the project got underway, we had a ritual of going around the table asking each administrator ''Can you accept an address of ...'')
3) Be open-minded about other people's mailers. It may not be just like a UNIX mailer, but it will likely get the job done.
4) It may take longer to document than to implement. Much longer. We found it helpful to draft an overview with sections on each domain which were updated as things changed. If your central mail gateway is a UNIX machine, it may cause your neighbors concern if they are not accustomed to UNIX. Be patient with them. The service you can provide will speak for itself.

Acknowledgments

SAS Institute Inc, a private corporation, develops, supports, and markets the computer software products that make up the SAS System, SYSTEM 2000 Data Management Software, NeoVisuals software, and the Institute's C compiler products.

Kudos to the mail group: Kevin Bond, Lisa Carter, Pete Cregger, Kent Fiala, Mark Fletcher, Diana Fox, John Hamlet, Marian Moore, Dave Tilley.

Helen Harrison is a UNIX System Administrator at SAS Institute, where she ministers to a diverse network of over 200 workstations, servers and mini-supercomputers. She holds a B.S. in Computer Science from Duke University. Reach Helen at SAS Institute; 1 SAS Campus Drive; Cary, NC 27512; or by e-mail at heh@unx.sas.com .

LISA IV
October 17-19, 1990
Colo. Springs, Colorado

Backup at Ohio State, Take 2

Steven M. Romig – The Ohio State University

ABSTRACT

In 1988 Elizabeth Zwicky presented a paper at LISA titled ''Backup at Ohio State'' (Workshop Proceedings, LISA II, September, 1988). We have been using essentially the same backup system since that time. Recently, the ever increasing size of our disk farm (currently roughly 22 gigabytes of backed up files), the increasing number of file servers involved in the backup process (37), and our continued desire to obtain painless yet reliable backups has lead us to design a new backups system, which uses essentially the same principles, with more general tools and with some interesting twists that may be useful to others. This paper describes our new system as it currently exists, and more importantly lists some of the tricks that we are employing.

Introduction

I have restored at least 150 file systems in the last 10 years. This has most often been in the course of doing carefully planned system upgrades, where the backups were meticulously done under controlled situations. Restoring file systems from these backup tapes has been easy and uneventful for the most part. I have also had the opportunity to restore file systems as a part of crash recovery. I have frequently encountered problems in restoring file systems in these cases, where the tapes are made by unknown people on unknown tape drives while the system was in an unknown state on an unknown date. Sometimes the tapes are unknown. Heisenberg uncertainty rears its ugly head...

As one might suspect, there are reasons for these problems. Tapes get lost. Operators mislabel tapes, or use the 2nd tape of one server's full save as part of the daily backup for another. We have occasionally had problems restoring files from tapes made while the file systems were active. Tracing the causal chain further, we find that it is hard to maintain a tape library of several thousand tapes, and to coordinate backups on 37 file servers, which contributes to the human error problems.

The old backup system consisted of a backup policy and a shell script used to actually perform the backups using dump. The policy was (in brief) to do full saves of each file system once a month, weekly saves (level 2 dumps in dump terminology) once a week, and daily saves (level 9) once a day. The backup script was somewhat user friendly, but had to be run on each file server by hand. The tape library and ''database'' (a log book) were maintained by hand.

The new system consists of several Perl scripts which allow the operators to set up a backup run which will cover the backups on several (possibly all) file servers. GNU tar is used as the archive program rather than dump, and we use 8 mm. tape drives rather than 1/2 inch. An on-line database is maintained with information about tape use and lots of log information, which allows us to write tools which greatly automate the tasks of tape management and error detection. As it turns out, there are a lot of other bonuses to the on-line database, which I will describe at the end.

Configuration File

One of the advantages of the new system is that the backup policies are not built into the software, for the most part. We use the familiar multiple level scheme used by Berkeley dump, where level 0 saves are full saves and the other levels are incremental against the most recent next lowest level save. A configuration file is used to name each level, to describe how often it should be done, and how long the tapes for that save should be retained before being reused. For example,

```
level 0 named full every 28 keep forever
level 2 named weekly every 7 keep 28
level 9 named daily every 1 keep 14
group one fish tree bird fruit pasta
group two oz pun monster mammal dinosaur
tape pun0 pun:/dev/nrst0
tape pun1 pun:/dev/nrst1
tape io1  io:/dev/nrst1
expire 1825 days 1000 uses
chains 2
```

This configuration file defines three backup levels named full, weekly and daily. Each *level* line identifies how often the backup should be done and how long the backup should be kept for. This sample specifies that a weekly (level 2) should be done every 7 days (measured since the most recent save whose level is less than or equal to 2) and kept for 28 days. The *group* lines define groups of hosts that can be referred to by the group name, which simplifies setting up a backup run for many hosts. The *tape* lines define aliases for tape host and device specifications. The *expire* entry defines the maximum age and number of uses that will be allowed for tapes in the tape library. Finally, the *chains* entry sets the number of backup chains to be maintained. A backup chain consists of a full save and its associated incrementals for some file system. There is a more complete discussion of backup chains in section titled ''Multiple Chains''.

The advantage to this is that we can readily change our procedures without changing the backup software. For example, I might want to keep the dailies longer at the cost of more tapes, or I might want to create extra chains, or add hosts to a group. Using a configuration file such as this means that we do not have to change the backup software just to effect some procedural change, which means that the software will be more stable and hopefully less prone to bugs.

Databases

Rather than describe the databases that the system maintains in gory detail, I will limit myself to a general overview. The system maintains three types of databases - one for tapes, one for backup runs, and one for backups.

The tape database contains information about the tape library, with one entry for each tape in the library. Tapes are identified by a unique number, which is written on a physical tape label, used as the index in the tape database, and which is written on a magnetic tape label at the start of the tape.

The magnetic label contains the tape number and information about the current contents of the tape, so if the tape somehow becomes dissociated from its physical label or the on-line databases are lost, we can still painstakingly recover everything (as long as we have the tapes). The operators also write most of the necessary information on the physical tape label itself in pencil. We also use the magnetic labels to check to see that the right tape is mounted in the right drive at critical times to avoid some of the problems of inadvertently destroying important backups.

Tapes are marked as *used* in the database if they contain data that is still *kept* (as defined in the config file for this level), otherwise they are marked as *free* and become part of the free tape pool. The database also contains information useful for expiring tapes by age or number of uses. We store the name of the last drive that the tape was written on so we can try to recover data from tapes written on drives with head alignment problems.

The run database contains information about each backup run. The user of the backup system does backups by initiating a backup run with the `all-backups` program. We make an entry in the run database for each backup run started. The entries in the run database contain all the specifications for that run (hosts, tape, drive, chain and level) and information that enables us to look up specific backup entries in the backup database that are part of this run.

Each save of every file system on every host gets an entry in the backup database. Each entry contains a list of the tape numbers and the file numbers on those tapes which are part of this save since each save can cover more than one tape. We also store the name of the compressed on-line copy of the save which is created under certain circumstances, and which can be useful in restoring files quickly without mounting tapes. A copy is made of any error messages that were written to stderr by the archiving program while the backup was being made, and we also maintain an on-line copy of the list of files stored in this save.

Programs

There are a plethora of programs associated with the new system. I will mention only the most important here.

The program that the operators use to initiate backup runs is called `all-backups`, so named because it arranges for a particular chain and level of backup to be run on a set of hosts, so it takes care of all of the backups for you. `All-backups` will ask the user for all the information that it needs, but you can also enter it all on the command line. The operator sets the chain, level, host list, tape drive, time and the options to be used in performing the run (for example, whether to do backups while rebooting, or whether to use the concurrent backup scheme). The tape can also be set, but usually `all-backups` will just pick a free tape from the tape pool. It does error checking where possible to ensure that you have the right tape in the drive and so on.

`All-backups` uses `do-backups` to do the backups on each of the hosts in its host list. `Do-backups` reads through `/etc/fstab` (or the equivalent) to determine what file systems on this host to backup. For each file system, it calls `backup`, which itself is a wrapper for GNU tar. `Backup` handles the database manipulations and maintains an `/etc/tardates` file which is used in conjunction with the incremental backup facility in GNU tar.

`Check-backups` is run on each file server every day through cron. It uses the config file and the `/etc/tardates` file to determine whether any backups for any of the file systems on this host are overdue. A mail message is sent to a list of people telling them whether the backups are ok, or listing the backups that are overdue.

`Clean-db` runs every day on the machine that hosts the database files. It removes old entries from the databases (according to the *keep* information in the configuration file), maintains the free tape pool and expires old or overused tapes.

The operators can review the database entries for a backup run with `show-run` which displays the relevant portions of the databases in a somewhat easy to read format. This is useful for watching the progress of a run or to check the results of one that has finished. Other programs provide for tape management such as adding and removing tapes to/from the library, in depth database reviewing, and so on.

Eventually there will be two restore programs: a simple version for doing full restores, and an interactive version for doing partial restores in an interactive manner. Both will use the various databases to determine which backups are required for the job, and they will both use the various tape and volume labels to ensure that the right tape and archive files are being restored from. The interactive version will use the on-line file catalogues to allow the user to traverse the virtual directory tree and create a list of files to be extracted in a fashion similar to the interactive restore mode on the BSD `restore` program.

Some Backup Tricks

I have incorporated some useful tricks in this backup system, which might prove to be useful in other backup systems as well. I will describe them here one by one.

Multiple Chains

One of the problems that I have frequently seen is that if you somehow lose tape 2 of the latest 3 tape full save set, you are up the creek. Sure, you could still restore from the previous full save and the incrementals from before the last missing full save and then from the most recent incrementals, but you can not do a full restore where files get renamed and deleted properly, and you are likely to have gaps in the record that were covered by the missing full save and so people will lose data and glare at you. Ouch.

A simple solution is to make one or more redundant *chains* of full saves and incrementals. Thinking in dump terminology, you might have two separate `/etc/dumpdates` files - one for dump chain A and the other for chain B. Set up dump to use `/etc/dumpdates.A` when you want to do a chain A dump, and `/etc/dumpdates.B` when you want to do a chain B dump. You would then have two independent sets of full saves and incrementals, and the loss of a single tape would not hurt you as much, since you can just restore from the other chain.

This might seem like a lot of extra work, but with the advent of high density media and unattended backups, it is not very painful to do. In our case, we have to schedule twice as many full saves (with associated downtime), but these will be done early in the morning when it is not as noticeable. And we have to do twice as many incrementals, but that just means that we load two tapes a day instead of one (all of our incrementals for a day fit on one tape). The cost of the extra media is not a burdensome problem since 8 mm. tapes are so inexpensive.

Concurrent Backups

One of the blessings of the high density media is that you can put a lot on one tape and reduce the number of tape switches. The downside is that you are introducing a point of serialization into your backup scheme which may drastically affect the amount of time it takes to backup all your systems since you are using one tape drive rather than several. It takes us about 2.5 hours to do the daily backups on 30+ hosts onto 5 1/2 inch tape drives. An older version of the new backup system took about 10 hours to do the dailies on 18 hosts on a single Exabyte tape, which we considered to be excessive.

The current version of the new backup system allows one to do the backups on many hosts in a concurrent fashion. Each host writes the saves for its file systems to a compressed file on one of its file systems if there is enough space or to a large empty file system available through NFS which is reserved for archives of this sort. Preference is given toward writing the saves locally, and to writing them onto file systems on different disk devices than the ones being backup up for improved performance. As these are finished, they are copied onto the tape and optionally deleted from the disk. If left on line they can be very convenient for restoring files. This scheme reduces the serializing effects of the single tape drive. Incremental backups on all our servers can now be done in about 3 hours.

Backups During Reboot

We have never been comfortable using dump to do our full saves while the file systems were active, and so in the past we have always taken our systems down to single user mode to perform the full saves. In the past this has always meant that the operator would have to take the system down, run the backup program, mount the tapes, and bring the system back up to multi-user mode when done, which of course requires that they be present while the backups are run.

Part of the point of using high density media is to reduce the amount of tedious work on the part of the operators. We have changed our `/etc/rc` file to enable us to initiate a backup run with quiescent file systems from cron and return to multi-user mode when done. The change is simple: insert the following into `/etc/rc` after the network and routing has been set up, but before other daemons that might cause file system activity have been started:

```
if [ -x /etc/backup ]; then
        sh /etc/backup
        rm -f /etc/backup
fi
```

If `all-backups` is told to run the backups while rebooting the hosts, it will create an `/etc/backup` file that contains the command to run `all-backups` with the appropriate command line to convey the information the user gave it. It can then execute `/etc/fastboot` directly or through `at` or `cron`. When the system reboots the backup will be done. When the backup is finished the boot process will continue and the system will (should) come back to life.

Rescheduling with At

`All-backups` can be told to setup a backup job to run at a specific time in the future, or it can be run now. If it is told to schedule a backup run, it creates an `at` job consisting of a command to run itself with the appropriate options set to convey the information gathered from the user, and dies. At will then restart the backup at the specified time. This can be very useful in conjunction with the ''backups during reboots'' feature so that one can schedule a full save for 3 A.M. when you leave for the day (whenever that is). `All-backups` checks that the right tape is in the drive before scheduling the `at` job, and again when the `at` job is started to catch any tape problems.

Backup Can be used by Real People Too

The current scheme started with the `backup` program, which provides a convenient wrapper for GNU tar that makes it easy to do full saves and incrementals of arbitrary directories. `Backup` can also be used by ordinary users without the database and with their own `tardates` file to create full saves and incrementals of their own directories whenever they want.

Redundant and Repetitive Information

The new system uses labels of various sorts to do as much error checking as possible. Each tape starts with a file that contains the tape number and other information such as the hosts, the chain and the level, and also has a physical label with the same number. The tape label is read by various programs to ensure that the right tape is being used. Each archive file on the tape contains a GNU tar volume header that identifies the host, file system, date and etcetera so that we can tell what is on the tape just by inspecting it. The volume header is also stored in the backup database so that the restore software can check to see that we are restoring data from the right archive on the tape.

Preventing Large Incrementals

One of the problems with backup systems on Berkeley style file systems is that if you do a full restore, the inode change times are changed, and subsequent incrementals will become very large since they are fooled into thinking that all the files have been changed. This backup system writes a file named `.chainN` (where N is the chain number) at the root of each file system just before a level 0 save is done. When incrementals are done another piece of code checks to see whether the inode change time on that file is later than the date of the last level 0 save for this chain. If it is, the `backup` program assumes that a full restore was done, and will print profuse error messages about the need for a full save on that file system. It then skips that file system. The `check-backups` program also checks for this condition, and will warn the operators that a full save should be done when it notices the need. Although this does have the unpleasant side effect of skipping the incremental save for that file system, we felt that it was reasonable for the backup system to force the users to do full saves where needed. This does not help at all in the situation where someone simply changes lots of data in the file system.

Conclusion and Future Plans

In general, we are happy with the new backup system. We should be switching to using it instead of our current system in the fall of 1990 (we are currently running both).

The restore software is currently being written, so we do not have any experience with it yet. We should be able to use the information from the database to make it easier for the operators to select tapes and restore files. The restore programs should be able to determine exactly which tapes to mount when, and it should be able to skip to the right archive file on the tapes in turn and extract the files with little or no help from the users. We should be able to make use of the various labels to reduce errors in restoring files, which can also be somewhat of a problem.

Eventually we will either enhance GNU tar or switch to another archive program. It would be nice to use Paul Placeway's fast rmt (''A Better Dump for BSD UNIX'', Workshop Proceedings, LISA III, September, 1989). GNU tar needs to be fixed so that it can correctly handle renaming directories when running in full restore mode. It would be nice to be able to get an on-line list of the contents of the tar archive using some other means than capturing the output of a verbose create. And somehow we would like to be able to get and save the size of the saves that are made, which we can use to estimate the size of saves we are about to do. It would also make it easy to study the rate of change of our various file systems over time. The concurrent backup scheme finds space by assuming that the save will take no more than about 25 megabytes, which is the largest compressed incremental save that we have seen on our file systems so far. If we can record the sizes of previous saves, then we can more accurately estimate the sizes of future saves since we should know roughly how much of a file system changes per day and how big yesterday's daily was, and then we can avoid the 25 Mb assumption.

At some point we will be adding a tape roll-over feature, where the user can specify an extra tape and drive to use if the first tape fills up. The system currently will notify the operator that it needs a new tape when it reaches end of tape, but this does not work so well when you are trying to run a backup in a fully unattended fashion at 4 A.M.

The concurrent backup scheme can leave its compressed incremental saves on disk after copying them to tape, where they can be useful in restoring files quickly. We might be able to make use of this to allow for some form of user initiated restores.

I would eventually like to add facilities to allow the backup system to determine its best guess at the best backup to do at the time so it will not be dependent on being run by knowledgeable users. It would also be nice to have facilities to analyze backup procedures and point out errors and to estimate the number of tapes that one will need to support a particular plan. Currently we rely heavily on human planning and scheduling.

Steve Romig is the software staff manager for the CIS Department at The Ohio State University. He received a B.S. degree in applied math from Carnegie Mellon University in 1982 and is slowly working toward an M.S. degree in computer science at Ohio State. His main professional interests are in simplifying and automating system administration tasks and in computer security. Reach him by U. S. Mail at Dept. of Computer and Information Science; 2036 Neil Avenue Mall; Columbus, OH 43210. Reach him electronically at romig@cis.ohio-state.edu. Telephonically, use 614-292-0915.

LISA IV
October 17-19, 1990
Colo. Springs, Colorado

The AFS 3.0 Backup System

Steve Lammert – Transarc Corporation

ABSTRACT

Much thought and effort has gone into providing mechanisms which augment or replace the standard UNIX dump(8) and friends. The design of the Transarc AFS 3.0 distributed file system — in particular, the *volume* representation of stored data — allows managers of AFS sites to perform file system backups in a simple, reliable, and transparent fashion. Snapshots of each volume can be taken and left on-line for reference by users; these snapshots or *clones* can then be written to tape without interrupting service or compromising data integrity.

Introduction

Backing up the file system is arguably the most important task facing a UNIX system administrator. In large heterogeneous computing environments, it can also be the most painful, both for administrators and users. The reasons for performing file system backups are well understood, as are most of the problems. The persistence of ''file system backup'' as a topic at USENIX conferences indicates both the universality and the intransigence of the problems.

The shortcomings of the standard UNIX backup tools have challenged system administrators and programmers to improve on the standard dump(8) and restore(8) utilities. These include front ends for dump [1], rewriting dump [3] and rdump [4], and throwing out the whole mess and starting over ([5], others). Generally, these efforts are built on top of the existing UNIX file system, and so use either the individual file or the entire physical partition as the fundamental unit of backup and restore.

Transarc's AFS 3.0 distributed file system [6] imposes an intermediate representation of file storage between the file and the partition. This intermediate object is called a *volume*[1], and is the focus of day-to-day administration of AFS file servers, including backup and restore operations. This paper outlines the backup facilities available to users and administrators of AFS 3.0 sites.

AFS Volumes

An AFS volume comprises a collection of files and directories, and forms a connected ''subtree'' of the file system. Typically, each user's home directory is contained in a separate volume, as are the system binary directories (e.g. /bin, /usr/ucb) for each supported hardware platform. These volumes are glued together at *mount points* to make up the AFS tree, much as physical partitions are mounted in a standard UNIX system to form a single file system image.

Volumes reside on UNIX workstations that have been designated as servers. A server has one or more physical disk partitions which have been set aside for volume storage. A number of volumes may be stored on the same server partition; a single volume may not span multiple partitions. Servers and their clients are grouped into administrative domains known as *cells*.

Several properties of AFS volumes enhance their ease of administration. Volumes do not have a fixed size, but grow or shrink on demand, subject to a dynamically adjustable per-volume quota and the available space on the partition. They may be moved from partition to partition, and from server to server within a cell, without changing their names and generally without the user being aware of the move. And they may be *cloned* — for moving, for read-only replication (identical copies of data on multiple partitions or servers), or for backup.

Volume Cloning

When a volume is cloned for backup, a *backup volume* is created which exists alongside the original ''read-write'' volume on the same partition. As created, the backup volume contains a reference to each and every file in the read-write volume; this reference is similar to a UNIX hard link. When a file is deleted from the read-write volume, the file is preserved by the reference in the backup volume. Later, when the backup volume is ''re-cloned'', the last reference to the file disappears and the space is reclaimed by the file system.

A backup volume takes up practically no space when it is created and can be generated in only a few seconds by a system administrator. During volume cloning, both the read-write volume and the backup volume are marked as ''busy''. In rare instances where a volume remains busy for too long, users may see the message ''afs: waiting for busy

[1]An extended form of the AFS 3.0 file system is incorporated in AFS 4.0, as selected by the Open Software Foundation for inclusion in its Distributed Computing Environment. In AFS 4.0, volumes are renamed to be *filesets*.

volume;'' these delays generally do not exceed 15 seconds, even for volumes of 50 MB or larger. Programs referencing data are unaware that the volume has been busy, except of course for the momentary delay.

Once created, the backup volume may be locked, dumped to tape, and deleted (or not) without further service interruptions. A common practice at AFS sites is to automatically generate a fresh backup copy of every user's volume early each morning.

On-line Recovery

Where a large percentage of data changes on a daily basis and can easily be regenerated (e.g. build trees for software developers), backup volumes are usually deleted after being dumped to tape — if indeed those volumes are worth backing up at all. However, most files on a typical UNIX system are relatively static, and backup volumes may retained for on-line access to the backed-up data.

A backup volume can be mounted in the AFS tree in the same way as read-write volumes. This provides immediate access to each user's backup volume, and eliminates a large percentage of file restoration requests.

For instance, my home directory is /afs/transarc.com/usr/shl, which is actually a mount point for the AFS volume named "usr.shl". Every morning at 5:30 AM, a cron job automatically makes a backup clone of all user volumes in our company cell. My backup volume is named "usr.shl.backup", and I usually mount it at ~shl/.Old.

```
% cd
% ls -a
 .Old/   .login  doc/    m.junk   mbox
 .cshrc  bin/    m.foo   m.trash  private/
% rm m*
% ls -a
 .Old/   .login  bin/    doc/     private/
 .cshrc
% ls -a .Old
 .Old/   .login  doc/    m.junk   mbox
 .cshrc  bin/    m.foo   m.trash  private/
% cp .Old/mbox mbox
% ls -a
 .Old/   .login  doc/    mbox     private/
 .cshrc  bin/
%
```

Example 1 – Recovering a deleted file

In the example, I have decided to remove the junk files in my home directory, all of which begin with "m." Unfortunately, I specified "m*" for deletion (rather then "m.*"), thus deleting my mbox. Fortunately, there is a backup volume mounted under ~shl/.Old, from which I can copy this morning's version of mbox back into my home directory.

The backup volume is a read-only clone. I can examine any file in the backup subtree and copy it back into my regular subtree, but I am prevented from deleting any file in the backup.

Of course, if I delete some files from my home directory and don't realize it until the next day (after 5:30 AM), those files will be wiped out by the automatic re-cloner, and I'll have to go and bother the operator for a file restore. Similarly, any modifications made and deleted on the same day are not protected, so that the user in Example 1 will have lost any mail processed that day. Still, the availability of the backup volume allows users to recover *on their own* from most common accidents.

Off-line Storage

In addition to the volume cloning mechanism, the AFS Backup System[2] provides a mechanism for storing and retrieving volumes on removable media. The system is composed of two programs which cooperate to select, transfer, and store volume data, and a database which keeps track of tape labels, tape drives, dump history, etc.

One or more machines in a cell are designated as *Backup machines*. A Backup machine need not be either an AFS server or a client, but it must be able to issue AFS RPCs and so must be a platform for which AFS is available (see the *Availability* section at the end of this paper). It also must have one or more locally attached tape drives.

A Backup machine runs both of the Backup programs: one instance of the command interpreter (called *backup*), and one or more instances of the tape coordinator (called *butc*), depending on the number of tape drives in use. The Backup database is stored on the local disk of this machine, so it should be located in a secure facility. Typically, an AFS file server doubles as a Backup machine and is already in a secure location.

The Backup Database

The Backup database contains information about tapes, volume sets, dump hierarchies, and dump sets. Tapes may be pre-labelled by the Backup system or labelled at first use, and are tracked by the system. During a dump or restore operation, the Backup system instructs the operator to mount specific tapes, and rejects any pre-labelled tape which does not match the mount request.

Volume sets are collections of volumes which have been grouped together by the administrator. A volume set includes volumes that are to be dumped together — on the same set of tapes, at the same

[2]This system is unrelated to the earlier CMU-developed system described in [2].

time, with the same frequency. A volume set is a list of volume entries, each of which describes a group of volumes; see Example 3 below.

Dump hierarchies are a set of logically related dump levels analogous to those in dump(8). Dump levels in the Backup system may be given names rather than numbers, which means that you can call the parent level (for full dumps) ''jul90'' rather than ''0'', and the subsidiary levels ''week1'', ''thursday'', etc. See Example 4 below.

Having defined one or more volume sets, and a dump hierarchy, the operator can then proceed to create *dump sets.* A dump set is the product of dumping a particular volume set at a given dump level; ''creating a dump set'' is equivalent to ''backing up the volumes in a volume set.''

To restore a volume, the operator requests a particular volume name for a particular date. Backup consults the database, decides which dump set is the correct one for the date requested, and instructs the operator to mount the appropriate tapes, one at a time, until the volume is restored. Volumes may be restored to their original name (overwriting any existing volume by that name), or to a different name for comparison.

Entire server partitions may also be restored. In this case, the operator requests a specific server and partition name, rather than a volume name. Backup then contacts an AFS volume location server, gets a list of the volumes which were on the affected partition, and instructs the operator to mount the appropriate tapes from the most recent dump sets until all of the lost volumes are restored.

Use of the Backup System

At Transarc, we currently have about 10 GB of data on three production servers and a varying number of test servers. Two of our servers double as backup machines. Each of the backup machines has an Exabyte 8 mm tape drive attached, capable of dumping up to 2 GB on a tape.

To perform a dump, the operator begins by invoking the Backup tape coordinator, *butc*. One instance of *butc* must be started for each tape drive on the Backup system; if there is more than one drive, a ''port offset'' is used to specify the drive to use. At our site, the operator uses an X terminal, and runs each *butc* process in a separate window.

```
% butc -help
Usage: butc [-port <port offset>]
            [-debuglevel <debug level>]
            [-help]
% butc
Tape Coordinator: Port offset 0 Debug level 1
```

Example 2 – Starting the butc process

Once the tape coordinators are running, the operator starts up the Backup command interpreter. The following examples demonstrate typical uses of the command interpreter.

```
# backup
backup> listvolsets
Volume set users:
  server .*, part .*, vols: usr.*backup

Volume set sources:
  server .*, part .*, vols: src.*backup
  server .*, part .*, vols: common.*backup
  server .*, part .*, vols: doc.*backup

Volume set sys.sun:
  server .*, part .*, vols: sun3_35.*backup
  server .*, part .*, vols: sun3_40.*backup
  server .*, part .*, vols: sun3x_40.*backup
  server .*, part .*, vols: sun4_40.*backup
  server .*, part .*, vols: sun4c_40.*backup

Volume set sys.dec
  server .*, part .*, vols: pmax_ul3.*backup
  server .*, part .*, vols: vax_ul3.*backup

Volume set sys.ibm:
  server .*, part .*, vols: rs_aix31.*backup
  server .*, part .*, vols: rt_aos4.*backup

Volume set sys.other:
  server .*, part .*, vols: hp300_70.*backup
  server .*, part .*, vols: next_10.*backup
backup>
```

Example 3 – Examining the volume sets

Example 3 shows the volume sets which have been defined in our development cell. There is one volume set containing all user volumes, one for sources, one each for Sun, DEC, and IBM binaries, and one for miscellaneous volumes. Each of these volume sets specifies a collection of volumes which currently weighs in at 0.5 GB - 1.5 GB, so that a complete dump will fit onto a single Exabyte tape. As the data in a volume set grows, Backup will ask for additional tapes where needed.

As previously mentioned, a volume set is a collection of volume entries. The syntax of a volume entry is:

```
server_name partition_name volume_name
```

Because volumes tend to move around from server to server, the first two fields typically are wildcarded with ''.*''. The last field may be specified with regular expressions. Thus, in the example above, the first volume set "users" specifies all user volumes on all server partitions.

Example 4 shows a dump hierarchy which provides for full monthly (archival) dumps, weekly incremental dumps relative to the monthly dump, and daily incrementals relative to the weekly tapes.

Soon, we plan to begin doing full weekly dumps as well as the archival monthly dumps.

The dump hierarchy is represented pictorially by indentation in the output from the ''listdumps'' command. You can think of the dump levels listed against the left margin as being the equivalent of dump(8) level 0, those at the first indentation as level 1 (or 5 or whatever) and those further in as level 2 (or 9, etc.). Additional levels may be specified as needed.

```
backup> listdumps
apr90
may90
jun90
jul90
    week1
        tue1
        wed1
        thu1
        fri1
    week2
        tue2
        wed2
        thu2
        fri2
backup>
```

Example 4 – Examining the dump hierarchy

A dump set is created by specifying a volume set and a dump level. Here the operator chooses to create a dump of all user volumes at the "wed2" level, which is an incremental based on the results of the "week2" and "jul90" dumps.

```
backup> dump -help
Usage: dump -volumeset <volume set name>
            -dump <dump level name>
           [-portoffset <TC port offset>]
           [-help ]
backup> dump users wed2
Starting dump of volume set 'users' (wed2)
```

Example 5 – Creating a dump set

At this point, the Backup system contacts the AFS volume location server, compiles a list of all volumes matching the wildcarded entries for the volume set ''users'', and contacts the tape coordinator at the specified port offset (defaulted in this case to port 0) to request that a specific tape be mounted. The operator mounts the tape, *butc* checks for the correct tape label, and the dump proceeds to completion.

Error recovery and logging

Sometimes a volume may be unavailable when the Backup system attempts to dump it. This could be due to a network partition, file server outage, or volume location server outage, or a problem with the volume itself. In this case, the Backup System omits the volume from the dump and continues to the next volume, rather than aborting or waiting for the volume to become available. The tape coordinator process notes the omission in its rolling output and in the log file.

For each volume, an incremental dump is always relative to the last successful dump rather than the last scheduled dump. Thus, if a volume is omitted from a full dump, the next incremental dump of that volume will include the entire volume. If a volume is omitted from an incremental dump, the next dump will be incremental with respect to the last actual dump, rather than to the most recent (missing) incremental.

Performance

The current release of the AFS Backup System transfers about 150 KB/sec from a DECstation 2100 to an Exabyte tape drive. About 85% of the data being written to the drive is from remote servers on a 10 MB Ethernet with about 90 active hosts; one of the servers is on the other side of a Cisco router. Running a second backup process to the other Exabyte on our SUN 4/260 server doesn't seem to affect the speed of the first drive at all. Although we generally spread our full backups over several days, with two drives we can easily do the full 10 GB in a single day.

Future work

An enhanced version of the AFS Backup System is now in development and will be shipped along with release 3.1 of AFS. The next release of the system will include:

- Cleaner error recovery on dumps and restores.
- Support for tape recycling and expiration.
- A tape database reconstruction facility.
- Various bug fixes.

A future release of the Backup system will introduce a *UBIK* -based Backup Database. UBIK databases are an integral part of AFS 3.0 and provide replicated authentication, protection, and volume location information to AFS servers and clients. The backup information will be similarly replicated and made available, so that all backup machines will share a single database. Information will no longer be stored on the local disk of the backup machine.

Transarc is actively seeking input on future releases of this software.

Acknowledgements

Thanks to Sailesh Chutani for implementing the first release of the Backup System, and to Pervaze Akhtar for continuing and enlarging upon Sailesh's work. Thanks also to Philip Lehman, Liz Hines, Ted Anderson, and Alfred for proofing this paper, both for grammar and content. Special thanks to Pat Barron for suggesting the method I used for getting the troff macros almost correct, despite our being a TeX-based environment.

AFS 3.0, which includes the Backup system described in this paper, is available from Transarc Corporation in Pittsburgh.

References

[1] Zwicky, Elizabeth, "Backup at Ohio State," *Proceedings of the USENIX Workshop on Large Installation Systems Administration*, USENIX Association, Berkeley CA, 1988.

[2] Hecht, Stephen, "Andrew Backup System," ibid.

[3] Placeway, Paul W., "A better dump for BSD UNIX," *Proceedings of the USENIX Workshop on Large Installation Systems Administration III*, USENIX Association, Berkeley, CA, 1989.

[4] Montgomery, Ken et al., "Filesystem Backups in a Heterogeneous Environment," ibid.

[5] Simicich, Nick, "YABS," ibid.

[6] Spector, Alfred Z. et al., "Uniting File Systems," *UNIX Review*, March 1989.

[7] *AFS 3.0 System Administrator's Guide*, FS-30-0-D102, Transarc Corporation, April 1990.

Steve Lammert is Facilities Coordinator for Transarc Corporation. He has ten years of experience in the management of computing facilities, including everything from writing user guides to stringing Ethernet cable. Prior to joining Transarc, he managed the Civil Engineering computing facilities at Carnegie Mellon University. Steve did his undergraduate work at Carnegie Mellon in the fields of electrical engineering, applied mathematics, and technical writing. He can be contacted at Transarc Corporation; 707 Grant Street; Pittsburgh, PA 15219; telephone (412) 338-4400; or via e-mail as shl@transarc.com.